University College London
Department of Computer Science

# Multi-objective Search-based Mobile Testing

*Ke Mao*

A dissertation submitted in partial fulfilment
of the requirements for the degree of
**Doctor of Philosophy**
of
**University College London**.

April 23, 2017

## Declaration

I, Ke Mao, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Some of the work presented in this thesis has previously been published or submitted during my PhD study at University College London:

[1] Ke Mao, Mark Harman and Yue Jia. Sapienz: Multi-objective automated testing for Android applications. In International Symposium on Software Testing and Analysis (ISSTA'16), 2016, pp. 94-105.

[2] Ke Mao, Licia Capra, Mark Harman and Yue Jia. A Survey of the Use of Crowdsourcing in Software Engineering. Journal of Systems and Software (JSS), vol. 126, pp. 57-84, 2016.

[3] Ke Mao, Mark Harman and Yue Jia. Robotic Testing of Mobile Apps for Truly Black-Box Automation. IEEE Software, vol. 34, pp. 11-16, 2017.

[4] Ke Mao. Towards Realistic Mobile Test Generation. ISSTA'16 Doctoral Symposium, 2016.

In addition, the following work has been published during my PhD programme. These papers are relevant to my research. However they do not form part of the thesis content.

[5] Ke Mao, Ye Yang, Qing Wang, Yue Jia and Mark Harman. Developer Recommendation for Crowdsourced Software Development Tasks. In Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE'15), 2015, pp. 187-198.

[6] Ke Mao, Qing Wang, Yue Jia and Mark Harman. PREM: Prestige Network Enhanced Developer-Task Matching for Crowdsourced Software Development. Technical Report RN/16/06, Department of Computer Science, University College London, 2016.

[7] Xinye Tang, Song Wang and Ke Mao. Will This Bug-fixing Change Break Regression Testing? In Proceedings of the 9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'15), 2015, pp. 1-10.

## Abstract

Despite the tremendous popularity of mobile applications, mobile testing still relies heavily on manual testing. This thesis presents mobile test automation approaches based on multi-objective search. We introduce three approaches: SAPIENZ (for native Android app testing), OCTOPUZ (for hybrid/web JavaScript app testing) and POLARIZ (for using crowdsourcing to support search-based mobile testing). These three approaches represent the primary scientific and technical contributions of the thesis.

Since crowdsourcing is, itself, an emerging research area, and less well understood than search-based software engineering, the thesis also provides the first comprehensive survey on the use of crowdsourcing in software testing (in particular) and in software engineering (more generally). This survey represents a secondary contribution.

SAPIENZ is an approach to Android testing that uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising their length, while simultaneously maximising their coverage and fault revelation. The results of empirical studies demonstrate that SAPIENZ significantly outperforms both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, on all three objectives. When applied to the top 1,000 Google Play apps, SAPIENZ found 558 unique, previously unknown crashes.

OCTOPUZ reuses the SAPIENZ multi-objective search approach for automated JavaScript testing, aiming to investigate whether it replicates the SAPIENZ' success on JavaScript testing. Experimental results on 10 real-world JavaScript apps provide evidence that OCTOPUZ significantly outperforms the state of the art (and current state of practice) in automated JavaScript testing.

POLARIZ is an approach that combines human (crowd) intelligence with machine (computational search) intelligence for mobile testing. It uses a platform that enables crowdsourced mobile testing from any source of app, via any terminal client, and by any crowd of workers. It generates replicable test scripts based on manual test traces produced by the crowd workforce, and automatically extracts from these test traces, motif events that can be used to improve search-based mobile testing approaches such as SAPIENZ.

## Impact Statement

Our multi-objective search-based SAPIENZ technique, can help Android developers reveal app crashes with minimised test paths, and maximised coverage that it can find. SAPIENZ is fully-automated, working on both emulators and real devices. It uses a novel motif gene representation and a hybrid exploration strategy to achieve high code coverage. Its multi-level instrumentation enables the testing of open-sourced and closed-sourced Android apps. When SAPIENZ finds a crash, it produces a detailed crash report and a crash video to help engineers locate the bug.

Our OCTOPUZ and POLARIZ techniques extend SAPIENZ. The former one makes the SAPIENZ technique applicable to the domain of JavaScript app testing, and the latter one enhances SAPIENZ performance by using crowdsourcing.

The work presented in this thesis reached some impact in academia. Our literature survey work on the use of crowdsourcing in software engineering has 43 citations. At the time of submitting this thesis, my work on search-based mobile testing and software crowdsourcing, have received 107 citations, according to Google Scholar.

Undoubtedly, the strongest evidence for impact comes from the 'MaJiCKe story'. MaJiCKe is an automated Android testing start-up company that has been constructed around the SAPIENZ approach. In February 2017, Facebook hired the team behind MaJiCKe (the author with two of his supervisors, Prof. Harman and Dr. Jia), demonstrating compelling evidence for the industrial relevance of the work reported in this thesis. The Facebook press release[1] reads:

> *"We're excited to announce that the team behind MaJiCKe will be joining us at Facebook in London. MaJiCKe has developed software that uses Search Based Software Engineering (SBSE) to help engineers find bugs while reducing the inefficiencies of writing test code. Their key product, Sapienz, is a multi-objective end-to-end testing system that automatically generates test sequences using SBSE to find crashes using the shortest path it can find."*
>
> – Facebook Academics

[1] `https://www.facebook.com/academics/photos/1326609704057100`

## Acknowledgements

I have been fortunate in studying a major that I truly enjoy and which never disappointed me. From Bachelor's to PhD, this is my 10th year majoring in computer science. Throughout this incredible journey, there are many people I would like to thank, without whom this thesis would not have been possible.

Special gratitude and appreciation go to my principle supervisor, Prof. Mark Harman. Mark supports me with invaluable guidance and freedom over the past several years. Even before I started my PhD, I was offered the opportunities to work with him on two papers and to present my work at a CREST Open Workshop. The detailed revision comments written by Mark contributed to each of my research submissions. The freedom provided by Mark allowed me to pursue my own research interests.

I also had a wonderful experience in working with my subsidiary supervisors Dr. Yue Jia and Prof. Licia Capra. I gratefully thank them for their insightful discussions and valuable feedback. I would like to acknowledge Dr. Yuta Maezawa and Prof. Shinichi Honiden, who offered me an excellent research collaboration experience at National Institute of Informatics. I want to thank Dr. Earl Barr for serving on my upgrade panel and thank Dr. Emmanuel Letier and Dr. Marouane Kessentini for serving on my thesis examination committee.

I am grateful to all my colleagues at CREST centre, for the numerous moments that I will cherish: the CREST 10th anniversary, COW workshops, citation lunches, etc. I want to thank Dr. William Langdon, for the research advice I received, and also for being the only officemate that I always met, no matter weekends or holidays. I am thankful to my PhD student colleagues for our years together: Afnan Alsubaihin, Alexandru Marginean, Bobby Bruce, Carlos Gavidia, Chaiyong Ragkhitwetsagul, DongGyun Han, Fan Wu, Lingbo Li, Matheus Paixao, William Martin, Zheng Gao.

I would like to thank my girlfriend Yiran, for her support and patience. I am grateful to all my friends for their company. I thank all my teachers, mentors, previous supervisors in schools, companies, universities and research institutes, without whom I cannot establish the foundation for this thesis. I sincerely thank my future colleagues at Facebook for their insights and support.

Finally, my deepest gratitude is to my family. I dedicate this thesis to my parents, for their continuous love and support throughout all my adventures.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Mobile applications (apps) have now overtaken the traditional desktop applications as the most widely used software system in digital media consumption [88]. The Apple and Google Play store have more than 4 million downloadable apps [350]. With the enhanced and engaging user experience, mobile apps provide a great number of services in business, entertainment and education sectors. This tremendous growth in mobile apps also provides a substantially profitable; but highly competitive, market for app developers. In order to attract more users, the development life cycles for mobile apps are usually short. A previous survey confirmed that app developers rarely follow a formal development process [380].

Despite the tremendous popularity of mobile apps, app quality problems remain prevalent. For example, app crashing is one of the most common issues complained about by mobile app users [193]. App crashes not only provide a bad user experience, but also have a negative impact on the overall rating of apps [193, 254]. To reveal mobile app quality problems before shipping the product to end users, testing, traditionally as a manual and expensive process, is necessary. According to a study on mobile app development [183], current mobile testing still relies heavily on manual testing, while the use of automated techniques remains rare [198]. Where fully-automated testing does occur, it typically uses random testing tools such as Google's Android Monkey [133], which is currently integrated with the Android system. Since this tool is so widely available and distributed, it is regarded as the current state of practice for automated

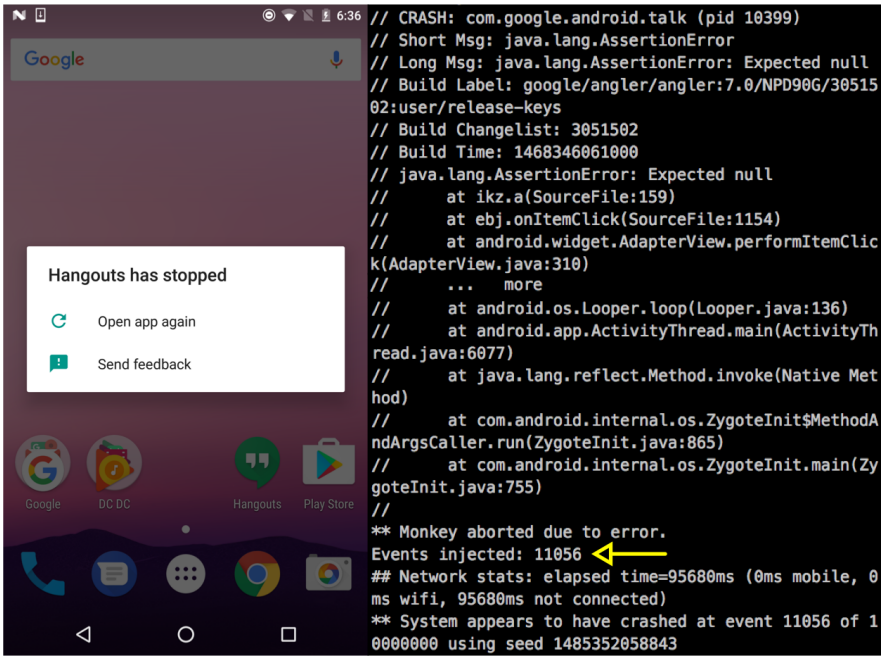



Figure 1.1: An example app crash found by Monkey. The length of the fault revealing test sequence is 11,056.

Android testing [240].

Although Monkey automates testing, it does so in a relatively unintelligent manner: generating sequences of events at random in the hope of exploring the app under test and revealing failures. It uses a standard, simple-but-effective, default test oracle [49] that regards any input that reveals a crash to be a fault-revealing test sequence. Automated testing clearly needs to find such faults, but it is no good if it does so with exceptionally long test sequences. An example crash, revealed by Android Monkey on a popular app 'Hangouts' (with 1-5 billion installs), is shown in Figure 1.1. Developers may reject such a long test sequence as being impractical for debugging and also unlikely to be reproduced by users: the longer the generated test sequence, the less likely it is to occur in practice. Therefore, a critical goal for automated testing is to find faults with the shortest possible test sequences, thereby making fault revelation more actionable to developers.

In the research area of automated mobile testing, while researchers have proposed many approaches for test generation, none of them simultaneously optimises multiple testing objectives. There is a lack of practical and comprehensive approach for system level test generation for testing apps with real-world complexity. For those available approaches

represented as the state of the art, a previous empirical evaluation study found that they do not show superiority over Android Monkey in terms of test coverage nor in terms of fault detection capability [84].

The key focus of this thesis is to advance the state of the art in automated mobile testing by proposing a new set of approaches and tools. We investigate the use of multi-objective search to generate short test sequences that maximise test coverage and fault revelation, for both Android native and JavaScript-based hybrid mobile apps. Furthermore, we seek to complement the multi-objective search by introducing collective crowd intelligence into computational search.

## 1.1 Research Problem and Motivation

We consider the research problem to be that of automated testing of mobile applications, where multiple conflicting test objectives are expected to be optimised simultaneously.

**Multi-objective Optimisation**: There are several competing objectives that testers may care about simultaneously, such as code coverage, test sequence length, number of crashes found, test realism, replicability, and execution time. Previous research mainly focuses on optimising code coverage and fault detection, which Moran et al. [273] found may not necessarily correlate with each other. How to trade off code coverage and fault detection capability against several other competing objectives such as test sequence length is challenging. Taking Monkey as an example, previous research generally found that it has a strong fault detection capability [84], while its generated test sequences are excessively long and the tests are usually not replicable. In this thesis, we consider three objectives, namely the code coverage, sequence length and the number of crashes found.

In the context of automated mobile testing, each executable test suite for the app under test (AUT) is termed as a *solution*. We denote the *solution* as $\vec{x}$, which is a vector of test cases (i.e., decision variables). We say solution $\vec{x_a}$ is *dominated* by solution $\vec{x_b}$

($\vec{x_a} \prec \vec{x_b}$) if and only if:

$$\begin{aligned} &\forall i = 1, ..., n,\ f_i(\vec{x_a}) \leq f_i(\vec{x_b}) \wedge \\ &\exists j = 1, ..., n,\ f_j(\vec{x_a}) < f_j(\vec{x_b}) \end{aligned} \tag{1.1}$$

A *Pareto optimal set* consists of all *Pareto optimal* solutions (belong to all solutions $X_t$), which is defined as:

$$P^* \triangleq \{\vec{x^*} \mid \nexists \vec{x} \in X_t, \vec{x} \prec \vec{x^*}\} \tag{1.2}$$

Further, a *Pareto optimal front* can be obtained by applying these Pareto optimal solutions:

$$PF^* \triangleq \{(f_1(\vec{x^*}), ..., f_n(\vec{x^*}))^T \mid \vec{x^*} \in P^*\} \tag{1.3}$$

**Problem Statement**: We formalise the problem of multi-objective automated testing as finding a *Pareto optimal front* $PF^*$, which consists of test suites that optimise $n$ ($n = 3$ in this thesis as described earlier) predefined objectives for testing the AUT. That is, we aim to find the test suite $\vec{x}$ that:

$$\begin{aligned} &\text{maximise } y = \vec{f}(\vec{x}) = (f_1(\vec{x}), ..., f_n(\vec{x}))^T \\ &\text{subject to } \begin{cases} g_i(\vec{x}) \geq 0, i = 1, ..., k \\ h_j(\vec{x}) = 0, j = 1, ..., l \end{cases} \end{aligned} \tag{1.4}$$

In above equation, $g_i$ and $h_j$ denote $k$ inequality constraints and $l$ equality constraints on test suite $\vec{x}$. We define a test suite that satisfies these constraints as an executable solution, and all *Pareto optimal* solutions must be executable.

The multi-objective optimisation problem is non-trivial and has been extensively studied in software engineering [110, 137, 301, 373, 405]. However, there is no readily available solution in the automated mobile testing domain. We investigate the potential solutions based on the Pareto-optimal Search Based Software Engineering (SBSE)

technique [143, 148]. We also aim to enhance the Pareto multi-objective search for mobile testing, by involving collective human intelligence in the search loop via crowdsourcing. The exploration of techniques that combine computational search intelligence and human intelligence for mobile testing is new and has not been considered in prior SBSE research.

The research work of this thesis on multi-objective search-based mobile testing is motivated by three sub-problems, regarding inadequate mobile test generation, the barrier in harnessing crowd intelligence to support automated mobile testing, and the lack of literature survey in incorporating crowd intelligence in software testing (and also more generally in software engineering).

**Inadequate Mobile Test Generation**: Mobile apps are unlike traditional enterprise software or web-based applications. They enable rich user interaction inputs such as gestures via touch screens and various signals via sensors (e.g., GPS, accelerometer, barometer, near-field communicator). They serve a wide range of users in heterogeneous and dynamic contexts such as geographical locations and networking infrastructures. These new features pose emerging challenges for mobile app testing.

Current mobile app testing in practice heavily relies on manual testing [183], while existing input generation techniques for testing mobile apps suffer from unrealistic (or unnatural) tests [234] and infeasible test coverage [84]. In this thesis, we provide a systematic approach to mitigate the inadequate test input generation issue by introducing multi-objective search and crowd intelligence.

**Bridging Crowdsourcing and Automated Mobile Testing**: Although crowdsourced testing has been widely practised in industry, the current state of practice usually simply dispatches the testing tasks to individual crowd testers and, thereby, only enables crowdsourced manual testing. Harnessing the crowd to support automated mobile testing is a non-trivial task due to several technical barriers in building an intermediary platform that can: manage the crowd and mobile devices, collect crowd test input accurately and model crowd intelligence as usable components for enhancing (search-based) mobile test automation. Also, it is challenging to find a way to effectively incorporate crowd intelligence into search-based automated mobile testing

approaches.

**Lack of Literature Survey**: Although there exist several surveys on search-based software testing [16, 22, 257], comparatively less is known about the overall development progress on using crowdsourcing to support software testing and other software engineering activities. There is no survey in this area. Recently, we have witnessed a dramatic rise in the work on Crowdsourced Software Engineering, yet many authors write that there is 'little work' on crowdsourcing for/in software engineering [332, 333, 351, 352, 353, 409]. In this thesis, we aim to fill the gap in the existing literature by providing a comprehensive survey of Crowdsourced Software Engineering. This survey covers crowdsourcing applications across many software engineering activities and serves as a spring-board for our use of crowdsourcing to support automated testing.

## 1.2 Objectives

The main objective of this thesis is to answer two high-level research questions:

RQ1: How can we enable fully-automated mobile testing by generating effective tests that optimise multiple objectives?

To drive the investigation multi-objective search for mobile testing, we aim to optimise three test objectives in this thesis, i.e., to maximise code coverage, fault revelation and meanwhile to minimise test sequence length. The coverage and length objectives can be measured based on the test inputs and instrumented subjects. Fault revelation requires generating automated/machine oracle, which is not the focus of this work. Instead, the aim is to study the test generator's fault revelation capability in terms of a general, yet important, automated implicit oracle [49], i.e., app failures/crashes.

Most mobile apps are either native (in Java for Android and in Objective-C/Swift for iOS) or hybrid (mostly in JavaScript). These two types of apps differ greatly, not only in their programming languages, but also in how they are coupled with the operating system (which directly impacts how their test automation systems should be designed).

RQ2: How can we effectively harness collective crowd intelligence to support search-based mobile testing?

In response to the challenge of combining human intelligence with computational search intelligence, we aim to introduce an approach that bridges the crowd and search-based mobile testing. The approach enables remote crowd testing of mobile apps and the automatic capture of manual test traces. Further, test trace mining is performed by the platform in order to learn from crowd intelligence. This necessitates an appropriate representation of crowd intelligence that can be subsequently used to enhance our multi-objective search-based mobile testing approaches.

In addition to the two research questions, the objective of addressing the lack of literature survey on software crowdsourcing are two-fold: First, we aim to provide a comprehensive survey of current research progress in the field of crowdsourced software testing, and more broadly, Crowdsourced Software Engineering. Second, to summarise the challenges for Crowdsourced Software Engineering and to reveal to what extent these challenges were addressed by existing work.

## 1.3 Contributions

By addressing the two high-level research questions, the main contributions of the thesis are:

- The first comprehensive literature survey of the use of crowdsourcing to support software testing and also other software engineering activities. The body of knowledge of this survey exposes trends, issues and opportunities for the emerging research area of Crowdsourced Software Engineering, which seeks augmentation of software engineering by incorporating collective human intelligence from the general public.
- The SAPIENZ approach: the first Pareto multi-objective approach to Android testing, combining techniques used for traditional automated testing, adapting and extending them for Android testing. The approach combines random fuzzing,

systematic and search-based exploration, string seeding and multi-level instrumentation, all of which have been extended to cater for, not only traditional white box coverage but also Android user interface (UI) coverage.

- The OCTOPUZ test generation approach, which reuses SAPIENZ' multi-objective search algorithm and explores the use of a comprehensive list of web UI events for effective and efficient practical JavaScript testing. It evolves efficient tests that optimise the three competing objectives of maximising coverage, maximising fault revelation, and minimising test sequence length.

- The POLARIZ approach for harnessing crowd intelligence to enhance search-based mobile testing techniques. The approach contains the design of a platform that enables and records crowdsourced mobile testing from any source of app under test, via any platform, by any crowd. The platform bridges the gap between automated mobile testing techniques and non-professional crowd testers. It further uses a novel motif extraction algorithm to mine the crowd intelligence underlying the automatically recorded test event traces. These mined motif events can be reused to improve existing search-based mobile testing approaches such as SAPIENZ.

- The first empirical study on using crowdsourcing to support mobile test automation. We demonstrate the usefulness of POLARIZ by posting 1,350 mobile testing tasks on Amazon Mechanical Turk to test 9 popular Google Play apps (each has at least one million user installs). A total of 434 crowd workers from 24 countries successfully participated the tasks with the help of the POLARIZ platform. The automatically learnt crowdsourced motif events were subsequently shown to be effective in improving SAPIENZ performance on coverage.

## 1.4 Thesis Outline

The rest of this thesis is organised as follows:

Chapter 2 (**Literature survey**) starts with a review on the state-of-the-art automated

mobile testing techniques. The chapter then moves on examining existing literature on leveraging crowdsourcing to support software engineering activities.

Chapter 3 (**Sapienz**) presents SAPIENZ, which provides a multi-objective search approach in Android app testing. The chapter fully describes the workflow and detailed algorithm of SAPIENZ. Evaluation experiments are performed on a dataset of 68 benchmark (open-sourced) F-droid apps and the top 1,000 most popular (closed-sourced) Google Play apps. Statistical analysis is also used to provide in-depth inferential statistical results on the performance of SAPIENZ compared to the state of the art and the state of practice.

Chapter 4 (**Octopuz**) discusses OCTOPUZ, the automated test generator that reuses SAPIENZ multi-objective search approach, but is redesigned and implemented for JavaScript app testing. An extension of a post test evaluator based on mutation testing is introduced. To examine whether OCTOPUZ can replicate the success of SAPIENZ, experiments are conducted on 10 real-world JavaScript apps, with results that demonstrate the superiority of OCTOPUZ over the state of practice and the state of the art in JavaScript testing.

Chapter 5 (**Polariz**) describes our POLARIZ approach, which is used to improve our SAPIENZ approach presented in Chapter 3. The chapter first describes the design of the POLARIZ and its crowdsourced motif extraction algorithm. Two empirical studies are then performed to show the usefulness of the POLARIZ platform for harnessing crowd workforce for remote mobile testing, and the effectiveness of POLARIZ motif extractor in mining crowd intelligence to enhance existing search-based mobile testing approach such as SAPIENZ.

Finally, Chapter 6 (**Future work**) closes the main body of this thesis work and shows our proof-of-concept future work on multi-objective robotic mobile testing. Chapter 7 (**Conclusions**) concludes.

# Chapter 2

# Literature Survey

This thesis is related to the previous work in mobile test automation techniques (such as random-based, model-based and search-based testing) for both Android and JavaScript mobile apps, and also the use of crowdsourcing in software engineering activities.

Although there already exist several literature surveys in search-based software testing [16, 22, 257] and also automated mobile testing [46, 153, 261], none of them introduced the use of emerging crowdsourcing model to support software testing. Indeed, there has been no survey on crowdsourced software engineering, more generally. This is a significant gap in the literature since crowdsourcing is not only important for our approach in this thesis, but also, as our survey reveals, it is increasingly important across the full range of software engineering activities. In the following sections, we first briefly summarise previous work on mobile test automation and related search-based software testing techniques. Subsequently, we present a comprehensive literature survey on the use of crowdsourcing to support software testing as well as other software engineering activities.

## 2.1 Automated Mobile Testing

Existing mobile apps fall into three types based on their targeted platforms and the used programming languages: native apps, web apps and hybrid apps. Native mobile

apps are written in a specific language for the targeted platform (e.g., Java for Android and Objective-C/Swift for iOS). Due to the open source nature of the Android platform, most existing research on automated mobile testing is conducted for Android applications. Web and hybrid apps are mostly written in JavaScript and HTML (and native code for hybrid apps), where the JavaScript components are most widely studied and tested as they usually control the main logic of the subject apps. In the following subsections, we present related work on automated testing for Android native apps and automated testing for JavaScript-based web/hybrid apps.

### 2.1.1 Automated Testing for Android Applications

Table 2.1 presents a brief survey of the characteristics of existing Android testing techniques and tools, which we briefly describe below.

The most closely related work employs search-based methods. Mahmood et al. introduced EvoDroid [240], the first search-based framework for Android testing. EvoDroid extracts the interface model (based on static analysis of manifest and XML configuration files) and a call graph model (based on code analysis by using MoDisco [9]). It uses these models to guide the computational search process. Unfortunately, its implementation is no longer publicly available.

Several previous approaches are based on random testing (fuzz testing), which inject arbitrary or contextual events into the subject apps. Monkey [133] is Google's official testing tool for Android apps, which is built into the Android platform, and therefore likely to be more widely used than any other automated testing tool for Android apps. Monkey generates (pseudo) random input events, which include both UI events, such as clicks and gestures, and system events such as screen-shot capture and volume-adjustment. Dynodroid [239] is a publicly available and open-source tool that extends pure random testing with two feedback directed biases: BiasedRandom, which uses context-adjusted weights for each event, and Frequency, which has a bias towards least recently used events. The implementation supports the generation of both UI and novel system events.

Table 2.1: At a glance: summary of existing tools and techniques for automated Android app testing ('OSS' and 'CSS' refer to Open-Source and Closed-Source Software used as evaluation subjects respectively)

| **Technique** | **Venue** | **Publicly Available** | **Box** | **Approach** | **Crash Report** | **Replay Scripts** | **Emulator / Real Device** | **Eval. Subjects** | | **Size** |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | **Type** | **OSS** | **CSS** |
| Monkey [133] | N/A | Yes | Black | Random-based | Text | No | Both | N/A | N/A | N/A |
| AndroidRipper [28] | ASE'12 | Yes | Black | Model-based | Text | No | Emulator | OSS | 1 | 0 |
| ACTEve [30] | FSE'12 | Yes | White | Program analysis | N/A | Yes | Emulator | OSS | 5 | 0 |
| $A^3E$ [41] | OOPSLA'13 | Partially | Grey | Model-based | N/A | Yes | Real device | CSS | 0 | 25 |
| SwiftHand [83] | OOPSLA'13 | Yes | Black | Model-based | N/A | No | Both | OSS | 10 | 0 |
| ORBIT [399] | FASE'13 | No | Grey | Model-based | N/A | No | Emulator | OSS | 8 | 0 |
| Dynodroid [239] | FSE'13 | Yes | Black | Random-based | Text, Image | Yes | Emulator | Both | 50 | 1,000 |
| PUMA [142] | MobiSys'14 | Yes | Black | Model-based | Text | Yes | Both | CSS | 0 | 3,600 |
| EvoDroid [240] | FSE'14 | No | White | Search-based | N/A | No | Emulator | OSS | 10 | 0 |
| SPAG-C [232] | TSE'15 | No | Black | Record-replay | N/A | Yes | Real device | Both | 3 | 2 |
| MonkeyLab [233] | MSR'15 | No | Black | Trace mining | N/A | Yes | Both | OSS | 5 | 0 |
| Thor [13] | ISSTA'15 | Yes | Black | Adverse conditions | Text, Image | Yes | Emulator | OSS | 4 | 0 |
| TrimDroid [268] | ICSE'16 | Yes | White | Program analysis | Text | Yes | Both | OSS | 14 | 0 |
| CrashScope [273] | ICST'16 | No | Black | Combination | Text, Image | Yes | Both | OSS | 61 | 0 |
| Sapienz | ISSTA'16 | Yes | Grey | Search-based | Text, Video | Yes | Both | Both | 78 | 1,000 |

GUI and model-based approaches are popular for testing Android apps [27, 28, 41, 83, 142, 399]. App event sequences can be generated from models, either manually constructed, or obtained from project artefacts, such as code or XML configuration files and UI execution states. For example, AndroidRipper [28] (subsequently MobiGUITAR [27]) builds a model using a depth-first search over the UI. Its implementation is publicly available however not open-sourced. $A^3E$ [41] consists of two app exploration strategies, the DFS strategy (like AndroidRipper) and a taint-targeted strategy which constructs a static activity transition graph. Although the tool is publicly available, the version does not support taint targeting. SwiftHand [83] dynamically builds a finite state machine model of the GUI, seeking to reduce restart time, while improving test coverage. ORBIT [399] is based on a combination of dynamic GUI crawling and static code analysis, using analysis to avoid generation of irrelevant UI events. PUMA [142] is a flexible framework for implementing various state-based test strategies.

Prior Android testing work also employs several other approaches, such as those that are program-analysis-based or reuse-based. ACTEve [30] is based on symbolic execution and concolic testing and supports the generation of both UI and system events. CrashScope [273] uses a combination of static and dynamic analyses to generate natural language crash descriptions with replicable test scripts. SPAG-C [232] implements a capture-reply based on image comparison of screen-shots to provide reusable and accurate test oracles, while Thor [13] makes use of existing test suites, seeking to expose them to adverse conditions. TrimDroid [268] is backed with program analysis by extracting interface activity transition and dependency models.

Collectively, these techniques cover several important test objectives, such as coverage, test sequence length, execution time, readability and replicability, yet none optimises these competing objectives simultaneously nor provides a set of optimal tradeoff solutions. Furthermore, many of these previously proposed techniques require detailed app information, such as source code [30, 240], general UI models [177] and interface and/or activity transition models [41, 240, 267, 268]. While any such additional information can help to guide test data generation, this additional information requirement can be an impediment to easy and widely-applicable automation.

Given the pressing need for *fully* automated Android testing, an approach to require only the binary executable is needed. Of the publicly available tools, Dynodroid and Monkey were found to perform best in the recent comprehensive study by Choudhary, Gorla and Orso [84]. Therefore, we regard these as denoting the state of the art and state of current practice, which we seek to improve by the introduction of an approach based on multi-objective search, which will be introduced in Chapter 3.

### 2.1.2 Automated Testing for JavaScript Applications

**Semi-automated JavaScript Testing.** There are several mature testing frameworks for JavaScript such as Jasmine[1], Mocha[2], and Karma[3], which are widely practised in industry. However all of these only automate the test execution phase and still require manual intervention for test case implementation and maintenance. Although these techniques are undoubtedly useful engineering frameworks and platforms for the development of test cases, effective and efficient testing requires automation, if it is to avoid tedium for the engineer and expense for his or her organisation. Frameworks such as these three leave the problem of achieving coverage and reducing test sequence length to human ingenuity.

**Search-based JavaScript test generation.** Search-based techniques have been widely used for test generation, especially for web applications [26, 38, 249, 250, 362, 381]. For those that specially target JavaScript-based web applications, Artemis [38] can be viewed as the first to treat the JavaScript testing problem as one involving a search space, although its search is a (feedback directed) random search, rather than one guided by fitness assessment, as more normally considered part of Search Based Software Engineering [146, 149]. Marchetto and Tonella used a hill climbing and simulated annealing approach for testing Ajax applications [249, 250]. Their search is guided by the diversity of the test suites and generated event sequences are of various lengths. The evaluation is performed on two open source Ajax applications with manually injected faults. The tool implementation of this work is not publicly available.

[1] http://jasmine.github.io
[2] http://mochajs.org
[3] http://karma-runner.github.io

**Mutation-based JavaScript testing.** Mutation testing is a fault-based technique to assess test-case adequacy. Although the faults injected by mutation testing are artificial in nature, there is evidence that indicates they can be representative of real faults [31, 185]. Since the 1970s [96, 141], various sets of mutation operators specific to programming languages and paradigms, such as Fortran, C, Java, C#, and AspectJ, have been proposed [178]. Several researchers have also defined effective sets of mutation operators specific to the .NET environment [242], PHP, JavaScript [338], HTML and JSPs [323].

Mirshokraie et al. developed Ajax-specific mutation operators based on available knowledge, such as JavaScript anti-patterns[4], provided by experienced programmers [265]. Nishiura et al. claimed that such 'best knowledge' might not cover mandatory features of Ajax technologies. Motivated by this observation, they conducted feature analysis of JavaScript programs used in web applications, proposing a tool called AjaxMutator with which they implemented a comprehensive set of mutation operators specific to Ajax technologies [292]; Olofsson reported that AjaxMutator might be the most mature tool for mutation testing of Ajax applications [298], thereby motivating our choice to use it in evaluating our multi-objective search-based JavaScript testing approach in Chapter 4.

Although a large body of work has been done on testing JavaScript-based applications, none of this optimises for several competing objectives such as code coverage, test sequence length and fault detection capability, simultaneously, for a mobile platform. This motivates us to conduct the work on multi-objective automated JavaScript testing in Chapter 4.

### 2.1.3 Related Search-based Software Testing Techniques

Search-Based Software Testing (SBST) is a branch of search-based software engineering. It uses metaheuristic search to generate software tests. Several previous surveys [16, 22, 151, 256, 257] suggest that, in the past decades, SBST has been extensively studied in the general software testing domain. These techniques may also have potential in

[4] `http://jaysoo.ca/2010/05/06/javascript-anti-patterns`

automated mobile testing, yet most of them have not been adapted and validated in testing real world mobile apps. In this subsection, we briefly introduce key search-based test generation techniques that relate to automated mobile testing. Based on the testing level of their generated tests, these techniques can be classified into unit, integration, and system test generation.

**Unit test generation.** Unit level tests focus on testing small pieces of code such as individual functions inside a class. Most previous search-based testing techniques are designed to generate unit tests [22]. Evosuite [117] is an open sourced SBST system for generate Java unit tests. It adopts a genetic algorithm to drive whole test suite evolution [115, 116], aiming to optimise test coverage. Its effectiveness in generating Java unit tests with high coverage has been demonstrated when testing 100 randomly sampled open source projects [113]. Godzilla [97] is a set of constraint-based unit test generation tools. It uses mutation analysis to generate tests with algebraic constraints, in order to find specific types of faults. Baars et al. [42] introduced symbolic search-based unit test generation. The approach incorporates symbolic execution information into the fitness function for covering branches more efficiently. Windisch et al. [388] investigated the use of Particle Swarm Optimization (PSO) in unit test generation. They demonstrated that PSO is an attractive alternative to popular genetic algorithm, due to it having fewer parameters and compelling performance in optimising coverage.

Unit tests are fast for execution, which generally makes the test generation process less expensive in time. They also provide well defined scope, which is friendly to developers for locating bugs. However, the interactions or dependencies among multiple classes/modules of the subject are not covered.

**Integration test generation.** Integration level tests cover the testing of inter-operations among multiple software modules. The concept of integration testing is vague and broad, which can vary from testing two classes where one depends on the other, to testing the integration with databases. Search-based integration test generation techniques have not been widely studied. Briand et al. [65] proposed an adapted genetic algorithm to generate integration test orders for object-oriented software. The proposed approach aims to reduce the complexity of stubbing during integration based

on inter-class coupling measurement.

Although generated integration tests cover multiple software modules interacting with each other, they do not test the subject software as a whole and operate its modules from the user's perspective. Those conditions that are unstated in the integration tests are neglected.

**System test generation.** System level tests treat the subject software as a black box and test it from the user's perspective. EXSYST [136, 329] is a search-based system test generation approach for Java GUI programs. It leverages random walk and dynamic analysis to guide the search toward optimising coverage. Similarly, Pidgin Crasher [92] uses evolutionary search to generate systems tests that optimise the number of crashes. The tool was built for the Pidgin instant messaging app specifically (but the technique also supports other Java programs that are based on the GTK+ framework). SWAT [26] is a tool based on a set of search-based testing algorithms for testing PHP web applications. Its evaluation on 10 real world web applications has demonstrated its effectiveness in reducing test effort and increasing coverage. Combinatorial interaction testing (CIT) has been demonstrated to be a useful technique for system testing [313]. Jia et al. [180] introduced a hyperheuristic algorithm to generate CIT tests. It learns different search strategies from multiple problem instances and its generalisation ability has been verified on a broad set of benchmarks.

Compared to unit and integration tests, system level tests are more natural (in simulating user interactions) and more comprehensive (in validating all layers of software). However, approaches to generating such systems tests also face additional challenges, especially for the mobile testing domain: First, executing system tests is an inherently time-consuming process. Executing system tests on mobile devices is generally much slower than on a desktop. Second, modern mobile apps accept various types of signals (multi-touch, acceleration, near-field communication, etc.) from device sensors that enable rich user inputs. The large search space corresponding to complex user interactions requires efficient search-based app exploration strategies. Third, mobile apps usually have user-centered design. Generating realistic system tests may be necessary to learn from human knowledge and to cater for multiple conflicting objectives that

affect test realism. Considering these challenges, it requires careful redesign and optimisation of existing search-based testing technique, even when they can be generalised to automated mobile testing in principle.

In this thesis, we focus on developing new techniques that optimise multiple conflicting objectives in system level mobile test generation. Specifically, we introduce the Sapienz, Octopuz and Polariz approaches in Chapter 3, Chapter 4 and Chapter 5, respectively. Sapienz adopts the whole test suite approach from the literature, but uses its own notion of coverage, based on Android events. Although previous work has sought short test sequence and high coverage, Sapienz is the first to combine these as a multi-objective search. Also, previous work has demonstrated the value of seeding search [26, 118], an idea incorporated into the Sapienz motifcore, which samples strings from the app to be re-used when needed by text-accepting GUI widgets.

## 2.2 Crowdsourced Software Engineering

The crowdsourcing model has been widely adopted in many computer science areas such as training data labelling in machine learning [404] and citizen sensing in pervasive computing [59]. In software engineering, many activities such as software testing inherently involve user participation and human intelligence, which may benefit from crowdsourcing. When seeking a survey of crowdsourcing to support the investigations performed in this thesis, we found there was no comprehensive survey of this important emerging area of software engineering, generally, and software test automation, specifically. We therefore set out to conduct such a survey to fill this gap in the literature. In this section, we present this comprehensive survey on the use of crowdsourcing in software testing and also other software engineering activities. This section is based on the author's survey paper published in the Journal of Systems and Software [247].

The term 'crowdsourcing' was jointly[5] coined by Howe and Robinson in 2006 [161]. According to the widely accepted definition presented in the article, crowdsourcing is the act of an organisation outsourcing their work to an undefined, networked labour

[5]Jeff Howe attributes the creation of the term to Mark Robinson and himself [162].

using an open call for participation.

Crowdsourced Software Engineering (CSE) derives from crowdsourcing. Using an open call, it recruits global online labour to work on various types of software engineering tasks, such as requirements extraction, design, coding and testing. This emerging model has been claimed to reduce time-to-market by increasing parallelism [201, 208, 351], and to lower costs and defect rates with flexible development capability [201]. Crowdsourced Software Engineering is implemented by many successful crowdsourcing platforms, such as TopCoder, AppStori, uTest, Mob4Hire and TestFlight.

The crowdsourcing model has been applied to a wide range of creative and design-based activities [25, 62, 74, 91, 294]. Crowdsourced Software Engineering has also rapidly gained increasing interest in both industrial and academic communities. Our pilot study of this survey reveals a dramatic rise in recent work on the use of crowdsourcing in software engineering, yet many authors claim that there is 'little work' on crowdsourcing for/in software engineering [332, 333, 409]. These authors can easily be forgiven for this misconception, since the field is growing quickly and touches many disparate aspects of software engineering, forming a literature that spreads over many different software engineering application areas. Although previous work demonstrates that crowdsourcing is a promising approach, it usually targets a specific activity/domain in software engineering. Little is yet known about the overall picture of what types of tasks have been applied in software engineering, which types are more suitable to be crowdsourced, and what the limitations of and issues for Crowdsourced Software Engineering are. This motivates the need for the comprehensive survey that we present here.

The purpose of our survey is two-fold: First, to provide a comprehensive survey of the current research progress on using crowdsourcing to support software testing and other software engineering activities. Second, to summarise the challenges for Crowdsourced Software Engineering and to reveal to what extent these challenges were addressed by existing work. Since this field is an emerging, fast-expanding area in software engineering yet to achieve full maturity, we aim to strive for breadth in this survey. The included literature may directly crowdsource software engineering tasks to the general

public, indirectly reuse existing crowdsourced knowledge, or propose a framework to enable the realisation/improvement of Crowdsourced Software Engineering.

The remaining parts of this survey are organised as follows. Section 2.2.1 introduces the methodology on literature search and selection, with detailed numbers for each step. Section 2.2.2 presents background information on Crowdsourced Software Engineering. Section 2.2.3 describes practical platforms for Crowdsourced Software Engineering, together with their typical processes and relevant case studies. Section 2.2.5 provides a finer-grained view of Crowdsourced Software Engineering based on their application domains in software development life-cycle. Sections 2.2.6 and 2.2.7 describe current issues, open problems and opportunities. Section 4.6 concludes.

### 2.2.1 Literature Search and Selection

The aim of conducting a comprehensive survey of all publications related to Crowdsourced Software Engineering necessitates a careful and thorough paper selection process. The process contains several steps which are described as follows:

To start with, we defined the inclusion criteria of the surveyed publications: The main criterion for including a paper in our survey is that the paper should describe research on crowdsourcing[6] that addresses at least one activity (directly or indirectly) involved in software engineering. A 'software engineering activity' can be any activity in the development, operation and maintenance of software, according to the IEEE Computer Society definition of software engineering [12]. Also, the literature must be presented in English as conference papers, journal papers, theses, technical reports or books.

We performed three types of searches on related publications published before April 2015:

- Online library search using seven major search engines: *ACM Digital Library*, *IEEE Xplore Digital Library*, *Springer Link Online Library*, *Wiley Online Library*,

[6]Note that since the crowdsourcing concept itself is expanding, its definition is still debated in the literature. Therefore, in order to ensure that our survey remains comprehensive, our inclusion criteria cover not only studies that meet our definition, but also those in which the authors claim to use crowdsourcing.

Table 2.2: Terms for online library search

| Category | Terms |
|---|---|
| General | (software crowdsourcing) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software engineering) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software development) |
| Domain | (crowd OR crowdsourcing OR crowdsourced) AND (software requirements) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software design) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software coding) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software testing) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software verification) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software evolution) |
| | (crowd OR crowdsourcing OR crowdsourced) AND (software maintenance) |

*Elsevier ScienceDirect*, *ProQuest Research Library* and *Google Scholar*. A list of search terms and their combinations we used are presented in Table 2.2. We searched each of the term combinations using exact match queries (e.g., "software crowdsourcing", "crowdsourced software development", "crowd testing", etc.) in both the meta-data and full-text (when available) of the publications.

- Issue-by-issue search of major conference proceedings and journals in software engineering from January 2006 to March 2015. This process was conducted manually to find those relevant papers that cannot be retrieved by the previous step. Our searched conference proceedings and journals are listed in Table 2.3.

- Reference search for identifying missed publications by going through citations from included ones (snowballing).

We conducted a screening process[7] to filter the collected literature by removing any that did not meet our inclusion criteria. We read the title and abstract (and the full text when necessary) of each publication carefully, applied the inclusion criteria and filtered out unrelated publications manually. We also performed a 'pseudo-crowdsourced' checking process for this survey. We contacted the authors (via email), to check whether

[7]The screening process is iterative, e.g., we also screened the publications suggested by the authors contacted and the anonymous reviewers.

Table 2.3: Selected conference proceedings and journals for manual search

| **Abbr.** | **Source** |
|---|---|
| ICSE | International Conference on Software Engineering |
| ESEC/FSE | European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering |
| OOPSLA | Conference on Object-Oriented Programming Systems, Languages, and Applications |
| ASE | International Conference on Automated Software Engineering |
| ISSTA | International Symposium on Software Testing and Analysis |
| ICST | International Conference on Software Testing, Verification and Validation |
| RE | International Requirements Engineering Conference |
| CSCW | Conference on Computer-Supported Cooperative Work and Social Computing |
| ISSC | International Symposium on Software Crowdsourcing |
| CSI-SE | International Workshop on Crowdsourcing in Software Engineering |
| TSE | Transactions on Software Engineering |
| TOSEM | Transactions on Software Engineering Methodology |
| IEEE SW | IEEE Software |
| IET | IET Software |
| IST | Information and Software Technology |
| JSS | Journal of Systems and Software |
| SQJ | Software Quality Journal |
| SPE | Software: Practice and Experience |

we had missed any important references and whether there was any inaccurate information regarding our description of their work. We then further revised the survey according to the authors' comments: we refined imprecise descriptions of their work and further included relevant papers that satisfied our inclusion criteria.

A detailed workflow of above steps and the number of resulting publications for each step are depicted in Figure 2.1. The initial type of search via online digital libraries produced 476 publications in total, where 132 of them are duplicated. The complementary issue-by-issue manual search led to another 53 unique papers. In total, 397 publications were reviewed by examining their titles and abstracts against our inclusion criteria. When the title and abstract of one publication did not give enough

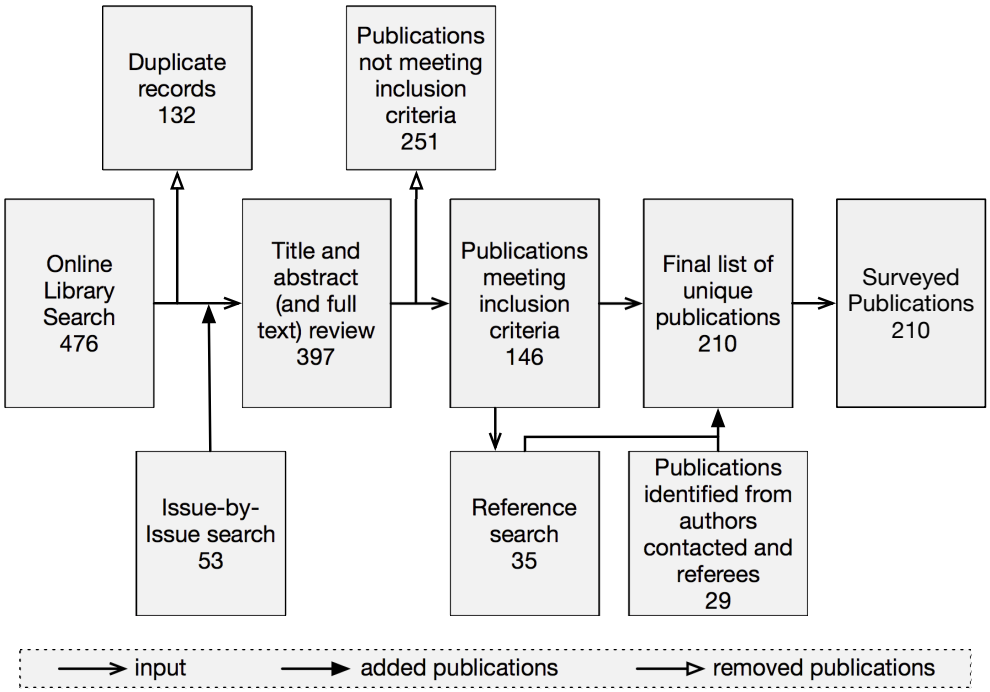



Figure 2.1: Results of literature search and selection on using crowdsourcing to support software engineering activities

information for making a decision, we further reviewed full-text of the paper. This step excluded 146 publications that did not meet the inclusion criteria. In addition, 35 publications were identified from reference lists of eligible publications. Regarding the 'pseudo-crowdsourced' step, for each included publication, we distributed the copy of this survey to at least one author. In total, we contacted 303 authors and received 83 replies. Twenty-two publications were suggested by these authors and another 7 were identified from the anonymous referees of this survey. Finally, a list of 210 unique publications remained, and were analysed in this survey. The growth trend in publications is presented in Figure 2.2. The distribution of these papers' publication types and a specific list of Master/PhD theses can be found in Figure 2.3 and Table 2.4, respectively. As can be seen, there is a noticeable rise in publications on Crowdsourced Software Engineering, resulting in a significant body of literature which we study in this survey.

We have built a repository which contains the meta-data of our collected papers. The meta-data includes the author, title, publication year, type and the conference proceeding/journal information of the paper. Based on this repository, we conducted our

analysis of the reviewed papers. This repository is publicly available online[8].

### 2.2.2 Definitions, Trends and Landscape

We first review definitions of crowdsourcing, before proceeding to the focus of Crowdsourced Software Engineering.

#### Crowdsourcing

The term 'Crowdsourcing' was first widely accepted in 2006. Jeff Howe used the term in his article 'The Rise of Crowdsourcing', which was published in *Wired* [161]. In a companion blog post [162] to this article, the term was defined explicitly:

> *"Crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call."*

According to this definition, the undefined, large networked workforce and the open call format are the two prerequisites of crowdsourcing. Howe argued that crowdsourced work can be done by cooperation or by sole individuals [161].

This idea echoes the earlier book 'The Wisdom of the Crowds' [174] and also finds some resonance in the principles of Open Source Software (OSS) development [199]. Indeed, although the term 'crowdsourcing' has attracted significant recent attention, the underlying concepts can be found in many earlier attempts to recruit a large suitably-skilled yet undefined workforce in an open call for a specific task in hand. For example, we might trace the origins of crowdsourcing back to the Longitude competition in 1714, when the British government announced an open call (with monetary prizes), for developing a method to measure a ship's longitude precisely [347].

Turning to online crowdsourcing, early Internet-based crowdsourcing activities can be found in 2001, when 'InnoCentive'[9] was funded by Eli Lilly to attract a crowd-based

[8] http://www.cs.ucl.ac.uk/staff/k.mao/cserep
[9] http://www.innocentive.com

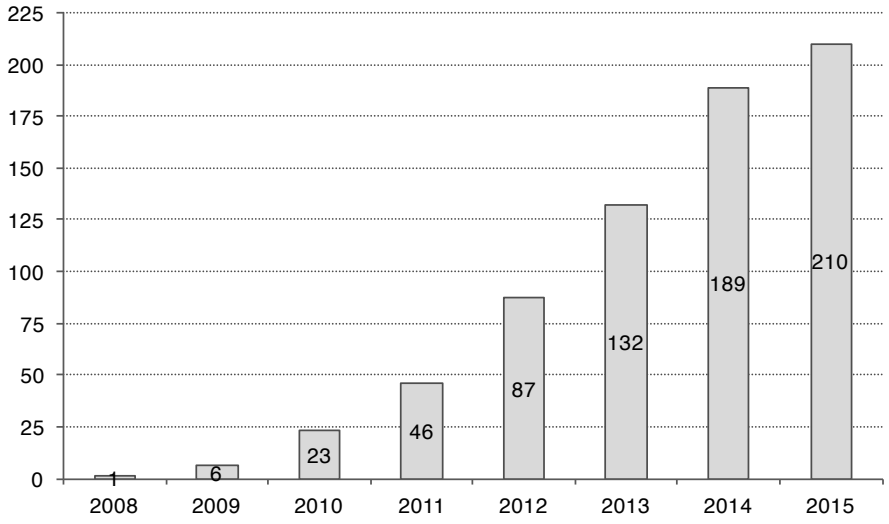



Figure 2.2: Cumulative growth of Crowdsourced Software Engineering papers published before April 2015

workforce from outside the company to assist with drug development. In the same year, the TopCoder platform was launched by Jack Hughes, as a marketplace using crowdsourcing for software development. To facilitate the online distributed software development activities, the TopCoder development method was proposed [165]. At the time of writing, TopCoder is the world's largest platform for Crowdsourced Software Engineering. By March 2015, its community of software engineers had numbered 750,000 and it had already awarded over $67,000,000 in monetary rewards for the Crowdsourced Software Engineering tasks it facilitated.

There are many other definitions of crowdsourcing, with subtle differences and nuances, which we review here. In Brabham's 2008 article [61], crowdsourcing is viewed as an online model for distributed production and problem-solving. The Wikipedia page on crowdsourcing[10] cites the definition which appeared in the Merriam-Webster dictionary in 2011[11]. It stresses the large group of workers and an online community, but drops any mention of 'undefined labour' and 'open call' format[12]. Estellés-Arolas et al. [106] collected 40 definitions from 32 articles published during 2006 to 2011, and proposed an integrated definition, which is compatible with the ones we have introduced and further specifies the mutual benefits between workers and requesters. Based on these previous definitions we can identify four common features that pertain to crowdsourcing: the

[10]http://en.wikipedia.org/wiki/crowdsourcing
[11]http://www.merriam-webster.com/info/newwords11.htm
[12]http://www.merriam-webster.com/dictionary/crowdsourcing

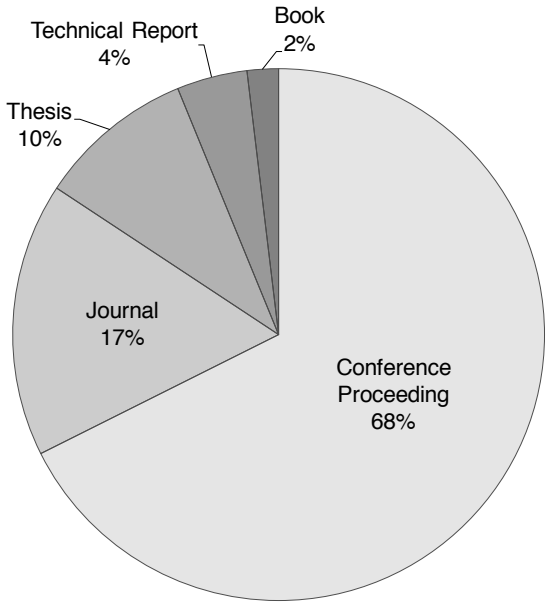



Figure 2.3: Publication type of surveyed papers

open access in production, the flexibility in workforce, the free will in participation and the mutual benefits among stakeholders.

The claimed benefits of crowdsourcing include easy access to a wide range of workers, diverse solutions, lower labour rates and reduced time-to-market. The granularity of crowdsourced tasks can be as finely grained as photo tagging or as coarsely grained as software development [197, 392]. A list of more than 160 crowdsourcing projects[13] has been compiled (using crowdsourcing to compile the list).

Crowdsourcing has been used extensively in various disciplines, such as protein structure prediction [91, 194], drug discovery [182, 294], transportation planning [62, 269], weather forecasting [74, 277], information retrieval [25, 215], and software engineering [63, 87, 332, 351, 354, 357], to which we now turn.

### Crowdsourced Software Engineering

We use the term 'Crowdsourced Software Engineering' to denote the application of crowdsourcing techniques to support software development (in its broadest sense). Some authors refer to this as 'Crowdsourced Software Development', 'Crowdsourcing Software Development' and 'Software Crowdsourcing' in previous studies [222, 324,

[13] http://en.wikipedia.org/wiki/list of crowdsourcing projects

Table 2.4: A list of master and PhD theses on Crowdsourced Software Engineering

| Year | Author | Degree | University | Title |
|---|---|---|---|---|
| 2010 | Lim [225] | PhD | University of New South Wales | Social Networks and Collaborative Filtering for Large-Scale Requirements Elicitation |
| 2011 | Manzoor [244] | Master | KTH - Royal Institute of Technology | A Crowdsourcing Framework for Software Localization |
| 2011 | Kallenbach [187] | Master | RWTH Aachen University | HelpMeOut - Crowdsourcing Suggestions to Programming Problems for Dynamic, Interpreted Languages |
| 2011 | Leone [218] | PhD | ETH Zurich - Swiss Federal Institute of Technology | Information Components as a Basis for Crowdsourced Information System Development |
| 2012 | Nag [281] | Master | Massachusetts Institute of Technology | Collabourative Competition for Crowdsourcing Spaceflight Software and STEM Education Using SPHERES Zero Robotics |
| 2012 | Saengkhattiya et al. [328] | Master | Lund University | Quality in Crowdsourcing - How Software Quality is Ensured in Software Crowdsourcing |
| 2012 | Gritti [135] | Master | Universitat Politécnica de Catalunya | Crowd Outsourcing for Software Localization |
| 2012 | Ponzanelli [319] | Master | University of Lugano | Exploiting Crowd Knowledge in the IDE |
| 2012 | Phair [315] | PhD | Colorado Technical University | Open Crowdsourcing: Leveraging Community Software Developers for IT Projects |
| 2012 | Bruch [67] | PhD | Technische Universität Darmstadt | IDE 2.0: Leveraging the Wisdom of the Software Engineering Crowds |
| 2012 | Goldman [129] | PhD | Massachusetts Institute of Technology | Software Development with Real-time Collabourative Editing |
| 2013 | Mijnhardt [262] | Master | Utrecht University | Crowdsourcing for Enterprise Software Localization |
| 2013 | Teinum [361] | Master | University of Agder | User Testing Tool: Towards a Tool for Crowdsource-Enabled Accessibility Evaluation of Websites |
| 2013 | Starov [349] | Master | East Carolina University | Cloud Platform for Research Crowdsourcing in Mobile Testing |
| 2013 | Chilana [80] | PhD | University of Washington | Supporting Users After Software Deployment through Selection-Based Crowdsourced Contextual Help |
| 2013 | Wightman [386] | PhD | Queen's University | Search Interfaces for Integrating Crowdsourced Code Snippets within Development Environments |
| 2013 | Xue [397] | PhD | University of Illinois at Urbana-Champaign | Using Redundancy to Improve Security and Testing |
| 2013 | Lin [230] | PhD | Carnegie Mellon University | Understanding and Capturing Peoples Mobile App Privacy Preferences |
| 2014 | Schiller [333] | PhD | University of Washington | Reducing the Usability Barrier to Specification and Verification |
| 2015 | Snijders [345] | Master | Utrecht University | Crowd-Centric Requirements Engineering: A Method based on Crowdsourcing and Gamification |

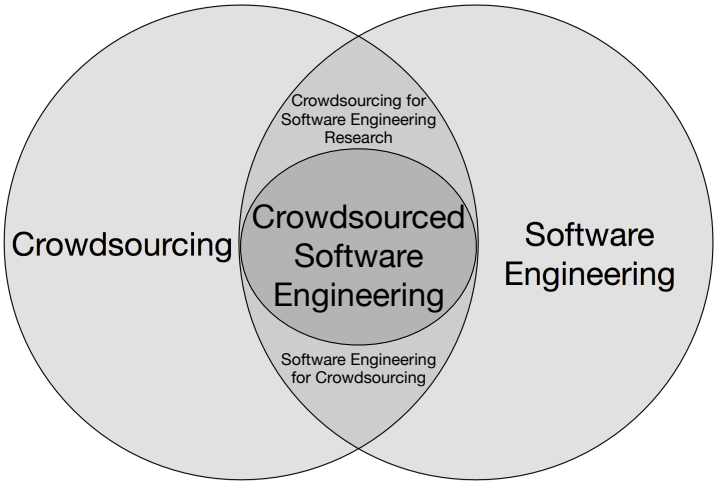



Figure 2.4: Crowdsourcing and software engineering

351, 358, 359, 368, 390, 391, 396]. However, we prefer the term 'Crowdsourced Software Engineering' since it emphasises *any* software engineering activity included, thereby encompassing activities that do not necessarily yield software in themselves. Example activities include project planning, requirements elicitation, security augmentation and test case refinement.

However, although our definition is inclusive of all software engineering activities, we wish to distinguish Crowdsourced Software Engineering from the research activities on software engineering that happen to be supported by Crowdsourcing (see Figure 5.13). Any research involving human subjects could potentially be supported by crowdsourcing, in the sense that the identification and recruitment of suitable human subjects for an experiment could be implemented using crowdsourcing techniques. In this application of crowdsourcing (to research studies), it is the identification of human subjects for experimentation that is important, rather than the particular research topic investigated.

If the research topic happens to be software engineering, then this work will be interesting to software engineers, but the principles and issues that arise will be more similar and relevant to those arising in other research involving human subjects. We call this application of crowdsourcing, 'crowdsourcing for software engineering research', to distinguish it from Crowdsourced Software Engineering. In this chapter, we comprehensively survey Crowdsourced Software Engineering. We do not claim to cover crowd-

Table 2.5: Cited crowdsourcing definitions

| **Def.** | None | Howe | Wiki | Own | Other |
|---|---|---|---|---|---|
| **Count** | 144 (69%) | 37 (18%) | 4 (2%) | 7 (3%) | 22 (10%) |

sourcing for software engineering research as comprehensively, although we do survey this topic for completeness. In addition, as shown in Figure 5.13, software engineering techniques can also be used to support the implementation of generic crowdsourcing, e.g., building a novel crowdsourcing system to support the crowdsourcing process [32]. This type of study is out of the scope of this survey.

Despite the wide usage of crowdsourcing in various software engineering tasks, the concept of Crowdsourced Software Engineering is seldom explicitly defined. According to our analysis (as shown in Table 2.5[14]), 69% of our surveyed papers use (or echo) the concept of crowdsourcing without citing any definition. For those papers that cite one or more definitions, the most widely cited is Howe's definition (18%). Out of all the 210 publications we reviewed, only two give an explicit definition of what it means for crowdsourcing to be applied specifically to software engineering activities [166, 351].

Stol and Fitzgerald's definition [351] refines Howe's crowdsourcing definition to the software development domain, requiring the undefined labour force to have requisite specialist knowledge. The definition from the 2013 Dagstuhl seminar on software crowdsourcing [166][15] is formalised as a Wikipedia page[16] on software crowdsourcing. It also specifies the tasks for software development, according to which the labour force remains unconstrained, yet the characteristic of a large potential workforce is not mentioned.

Since Howe's definition is the most widely accepted crowdsourcing definition in the papers we surveyed, we choose to define Crowdsourced Software Engineering (CSE) simply as an instantiation of Howe's definition, as follows:

> *Crowdsourced Software Engineering is the act of undertaking any external software engineering tasks by an undefined, potentially large group of online workers in an open call format.*

[14] One single paper may cite multiple definitions.
[15] Subsequently published as a book: `http://www.springer.com/gb/book/9783662470107`
[16] http://en.wikipedia.org/wiki/crowdsourcing_software_development

Note that in this survey, we also include the work that uses crowd knowledge to support software engineering activities, which can be regarded as a form of indirect use of crowdsourcing. For example, those 'crowd debugging' [76, 154, 276] studies on collecting and mining crowdsourced knowledge. We observed that the software engineering community generally considers this as an instance of using crowdsourcing to support software engineering, because the knowledge used was gathered from 'crowd workers', as defined in Howe's definition.

Crowdsourced Software Engineering generally involves three types of actors (or stakeholders): Employers (aka requesters), who have software development work that needs to be done; workers, who participate in developing software and platforms, which provide an online marketplace within which requesters and workers can meet. Figure 2.5 briefly depicts these three types of actors and the general process for Crowdsourced Software Engineering. More detailed background knowledge on the Crowdsourced Software Engineering actors, principles, process and frameworks can be found in the recent book by Li et al. [222].

**Claimed Advantages and Growth Trends**

Crowdsourced Software Engineering has several potential advantages compared to traditional software development methods. Crowdsourcing may help software development organisations integrate elastic, external human resources to reduce cost from internal employment, and exploit the distributed production model to speed up the development process.

For example, compared to conventional software development, the practice of TopCoder's crowdsourced software development was claimed to exhibit the ability to deliver customer requested software assets with a lower defect rate at lower cost in less time [201]. TopCoder claimed that their crowdsourced development was capable of reducing cost by 30%-80% when compared with in-house development or outsourcing [236]. Furthermore, in the TopCoder American Online case study [201], the defect rate was reported to be 5 to 8 times lower compared with traditional software development practices.

In another study published in *Nature Biotechnology* [202], Harvard Medical School adopted Crowdsourced Software Engineering to improve DNA sequence gapped alignment search algorithms. With a development period of two weeks, the best crowd solution was able to achieve higher accuracy and three orders of magnitude performance improvement in speed, compared to the US National Institutes of Health's MegaBLAST.

The increasing popularity of Crowdsourced Software Engineering revolves around its appeal to three different related stakeholders:

*1) Requesters:* Crowdsourced Software Engineering is becoming widely accepted by companies and organisations, from the military domain and academic institutions to large IT companies. DARPA created Crowdsourced Formal Verification (CSFV) program[17] for software formal verification and launched the Verigames website[18] to facilitate the practice[19]. NASA and Harvard business school established the NASA Tournament Laboratory for crowdsourcing software solutions for NASA systems[20]. Microsoft crowdsourced partial software development activities in Office 2010[21], Windows 8.1[22] and Windows 10[23].

*2) Workers:* Based on an industrial report from Massolution [255], the number of workers engaged in software crowdsourcing increased by 151% in the year of 2011.

*3) Platforms:* There is a growing number of crowdsourcing platforms built for software development domain, such as AppStori and Mob4Hire. These commercial platforms will be described in more detail in Section 2.2.3.

The flourishing Crowdsourced Software Engineering landscape is also revealed by the increasing number of relevant publications published in recent years, as shown in Figure 2.2. Crowdsourced Software Engineering is also proving to be an attractive topic for

[17] http://www.darpa.mil/program/crowd-sourced-formal-verification
[18] http://www.verigames.com
[19] http://www.verigames.com
[20] http://www.nasa.gov/sites/default/files/files/ntl-overview-sheet.pdf
[21] http://www.wired.com/2010/06/microsoft-office-2010
[22] http://www.forbes.com/sites/andygreenberg/2013/06/19/microsoft-finally-offers-to-pay-hackers-for-security-bugs-with-100000-bounty
[23] http://thetechieguy.com/how-microsoft-is-cleverly-crowdsourcing-windows-10-development-from-its-customers

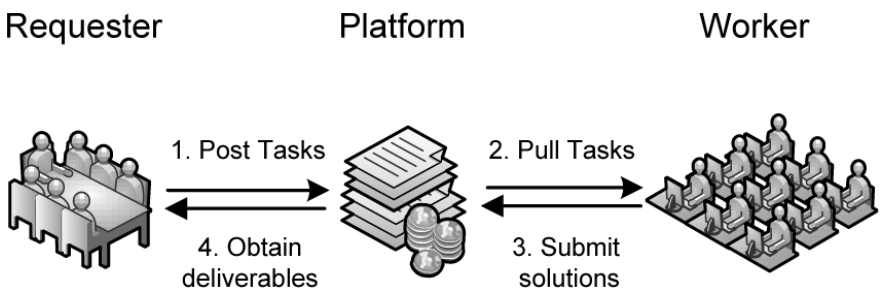



Figure 2.5: Actors in Crowdsourced Software Engineering

student dissertations. Specifically, 20 out of the total 210 publications are Master/PhD theses. A detailed list of these theses can be found in Table 2.4.

**Research Topics**

To classify the papers, we first carefully analysed the 210 papers we collected, revealing four top-level categories based on their study types: Study of Practice, Theories and Models, Applications to Software Engineering and Evaluations of Software Engineering Research. We referred to the ACM Computing Classification System[24], the IEEE Taxonomy of Software Engineering Standards[25] and the 2014 IEEE Keywords Taxonomy[26] to formulate sub-categories for each of these four top-level categories. Specifically, for applications to software engineering, we created sub-categories based on different stages of software development life-cycle addressed by the applications. A detailed taxonomy of Crowdsourced Software Engineering research is given in Figure 2.6.

We manually classified the collected papers and assigned them to each of the categories. The classification results were cross-checked by three authors, reaching an average percentage agreement of 91.2%. The distribution of the literature over the research topics is shown in Figure 2.7. The most prominent class is Applications to Software Engineering (64%), followed by theoretical studies (19%) and practical studies (14%). A few studies (3% in our collection of papers) employed crowdsourcing to evaluate software engineering research. This type of publication may not use crowdsourcing-related keywords in their meta information. We performed extra manual retrievals for related research. Nevertheless, there may be more papers which fall into this category yet which remain uncovered in our survey; this category is not the focus of our survey.

[24]http://www.acm.org/about/class/class/2012
[25]http://ieeexplore.ieee.org/servlet/opac?punumber=2601
[26]http://www.ieee.org/documents/taxonomy_v101.pdf

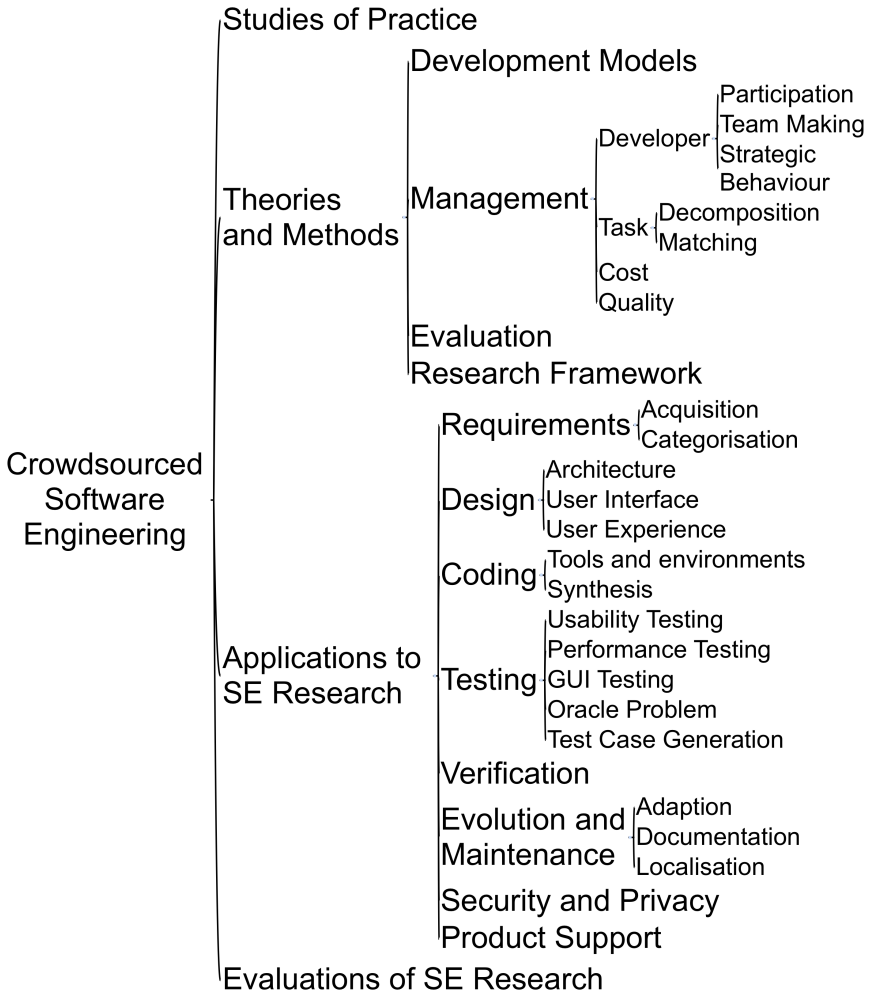



Figure 2.6: Taxonomy of research on Crowdsourced Software Engineering

**CSE Research Landscape** We present the CSE research landscape from two views: a specific view of the Software Development Life Cycle (SDLC) and a general view of the problem-solving process. The former view illustrates a picture of the activities that have been covered in published literature on applying crowdsourcing to software engineering. The latter points out a series of questions and corresponding variables on why and how to use crowdsourcing to tackle a problem in software engineering. These questions have not been widely discussed in a software engineering context. The latter view also identifies several issues remaining open to CSE researchers (who wish to design new crowdsourcing-based approaches) and practitioners (who intend to adopt existing crowdsourcing-based approaches), thus reflecting the landscape of future CSE research topics from a process perspective.

In Figure 2.8, we illustrate the domains in which crowdsourcing has been integrated

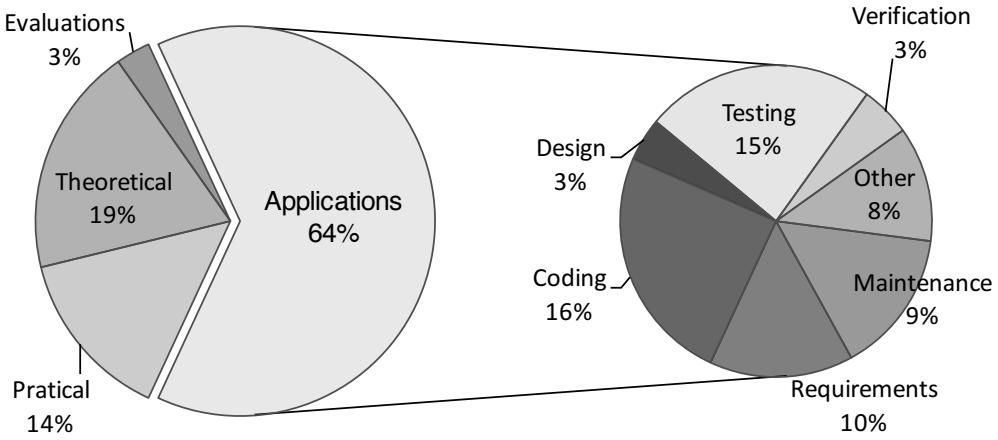



Figure 2.7: Research topic distribution

into the SDLC. Crowdsourcing can be used to support activities involved in the planning, analysis, design, implementation and maintenance phases of a software system. For each specific activity in the SDLC, we also provide an example reference on realising this integration. The summary of the papers that apply crowdsourcing to software engineering activities and their mappings to the SDLC are discussed in detail in Section 2.2.5.

When adopting crowdsourcing to support a certain activity in an SDLC phase, it requires a general problem-solving process which characterises the steps towards approaching and rationally solving a problem. We illustrate the CSE research landscape from such a general perspective of problem-solving process, as shown in Figure 2.9, where important questions and aspects regarding a series of stages in realising CSE are summarised. The stages follow Simon's problem-solving model [344], which consists of two phases: a decision phase and an implementation phase. Since Simon's problem-solving model has been adapted in several ways, each phase may vary in stages. Our decision phase contains the three typical stages: intelligence, design and choice. While the implementation phase includes an implementation stage and a following review stage.

*Intelligence (why)*: In the intelligence stage, the problem is defined and the requester should justify the motivation for adopting CSE. What are the potential advantages and disadvantages? Previous research on CSE usually argues that the cost, efficiency and scalability are the benefits. Meanwhile, the potential issues on intellectual property

and the quality of the crowd work may need to be considered.

*Design (what)*: The design stage relates to the development of alternative solutions. This phase may require research into the potential options. When mapping to the CSE stage, the requester should think about, what is being crowdsourced? What is the granularity of the task? Is it a micro-task such as a requirement classification or a macro task such as a software component development? Does the task require expert knowledge such as programming? What are the incentives for a certain type of crowd? What is the expected size of the crowd?

*Choice (which)*: In the choice stage, alternative solutions are evaluated. The output of this stage is a decision that can be implemented. To make such a decision usually requires additional information, which has not been collected in the previous design stage. Other choices would also be considered, such as the quality assurance strategy and the type of open call to be adopted.

*Implementation (how)*: The implementation stage is where the decision is finally carried out. Both CSE researchers and practitioners need to deal with a series of questions to implement the crowdsourcing activity. For example, what intermediary platform should be used in order to accomplish the work? Subsequently, how to manage the tasks and workers?

*Review (outcome)*: The final review stage evaluates the implementation's outcome. If the outcome is a success, the product may be further validated to check whether it satisfies users' requirements. If failed, lessons should be learned and a new iteration of the problem-solving process is expected. For CSE, the requester may need to think about how to aggregate and validate the crowd solutions. If the use of crowdsourcing in supporting software engineering failed, what is the barrier to a successful CSE solution?

The above process model may provide guidance to CSE researchers and practitioners for realising CSE. Several of the questions underlying each stage remain open problems, pointing to important research questions. Very few previous studies consider the overall process of CSE at a macro level (such as the one presented in Figure 2.9). Instead, where the process studies were conducted, they usually focused on the problems pertained to

a specific stage.

In the *why* stage, many previous studies report on the benefits of CSE in terms of low cost, fast delivery and high flexibility [53, 201, 208, 222, 288, 325, 334], but there is a lack of comparative studies on validating these benefits by comparing crowdsourcing-based development with traditional in-house or outsourced development. A few other authors, see for example [207, 351, 353], highlight the key concerns associated with CSE. In the *what* stage, research topics of 'what to crowdsource', 'who is the crowd', and 'what is the motivation' have been briefly touched upon. Stol and Fitzgerald [351] point out that self-contained, less complex software tasks are more suitable to be crowdsourced, according to an in-depth study on TopCoder. Schiller and Ernst [332], and Pastore et al. [308] discuss using ad-hoc versus contract workers and professional versus generic workers, respectively. A detailed discussion of the motivation and composition of the crowd will be presented in Sections 2.2.6.In the following *which*, *how* and *outcome* stages, it is surprising to see that not much research work has been done so far. For 'which CSE model', a few process models have been proposed, which will be introduced in Section 2.2.6. For 'which quality assurance strategy', relevant studies will be addressed in Section 2.2.6. Regarding the *how* stage, the challenges of task decomposition and crowd coordination will be discussed in Sections 2.2.6 and 2.2.6.

In contrast to the lack of investigation of the CSE process, most studies reviewed in this survey integrate crowdsourcing into their approaches to support activities in the SDLC. In later sections, we first summarise these papers (Section 2.2.5) and then turn back to the CSE process, discussing the issues and open problems studied in previous work (Section 2.2.6), and opportunities for future research (Section 2.2.7).

### 2.2.3 Crowdsourcing Practice in Software Engineering

In this section, we describe the most prevalent crowdsourcing platforms together with typical crowdsourced development processes for software engineering. Since most case studies we collected were based on one (or several) of these commercial platforms, in the second part of this section, we present relevant case studies on the practice of Crowdsourced Software Engineering.

Table 2.6: A list of commercial platforms for Crowdsourced Software Engineering

| **Platform** | **Since** | **URL** | **Task Domain** | **Open Call Form** |
|---|---|---|---|---|
| TopCoder | 2001 | topcoder.com | Software Development | Online Competition |
| GetACoder | 2004 | getacoder.com | Software Development | Online Bidding |
| AppStori | 2013 | appstori.com | Mobile App Development | Crowd Funding, Online Recruiting |
| Bountify | 2013 | bountify.co | Small Coding Tasks | Online Competition |
| uTest | 2007 | utest.com | Software Testing | On-demand Matching, Online Competition |
| Passbrains | 2012 | passbrains.com | Software Testing | On-demand Matching |
| 99Tests | 2010 | 99tests.com | Software Testing | On-demand Matching |
| TestBirds | 2011 | testbirds.com | Software Testing | On-demand Matching |
| Testbats | 2013 | testbats.com | Software Testing | On-demand Matching |
| Pay4Bugs | 2009 | pay4bugs.com | Software Testing | On-demand Matching |
| CrowdTesters | 2014 | crowdtesters.com.au | Software Testing | On-demand Matching |
| TestFlight | 2010 | testflightapp.com | Mobile App Testing | On-demand Matching |
| Mob4hire | 2008 | mob4hire.com | Mobile App Testing | Online Bidding |
| Testin | 2011 | itestin.com | Mobile App Testing | On-demand Matching |
| Ce.WooYun | 2012 | ce.wooyun.org | Software Security Testing | On-demand Matching |
| Bugcrowd | 2012 | bugcrowd.com | Software Security Testing | Online Competition |

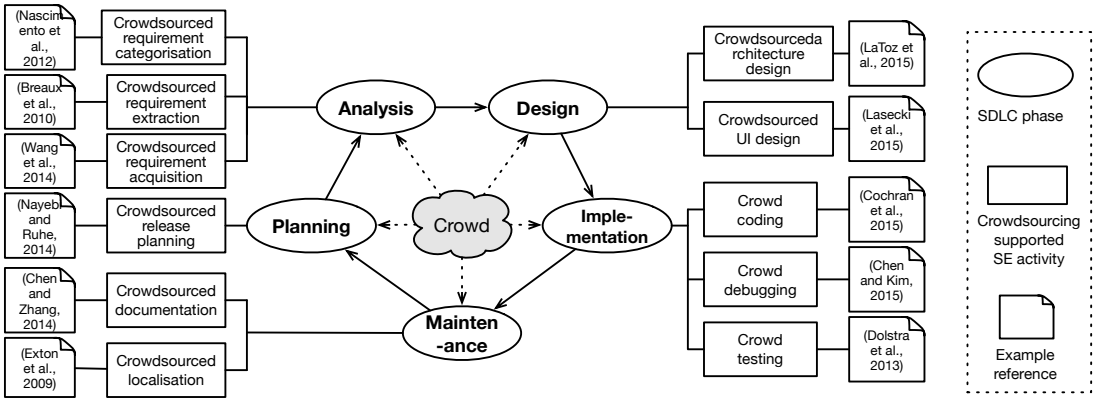



Figure 2.8: Integration of crowdsourcing into the Software Development Life Cycle (SDLC)

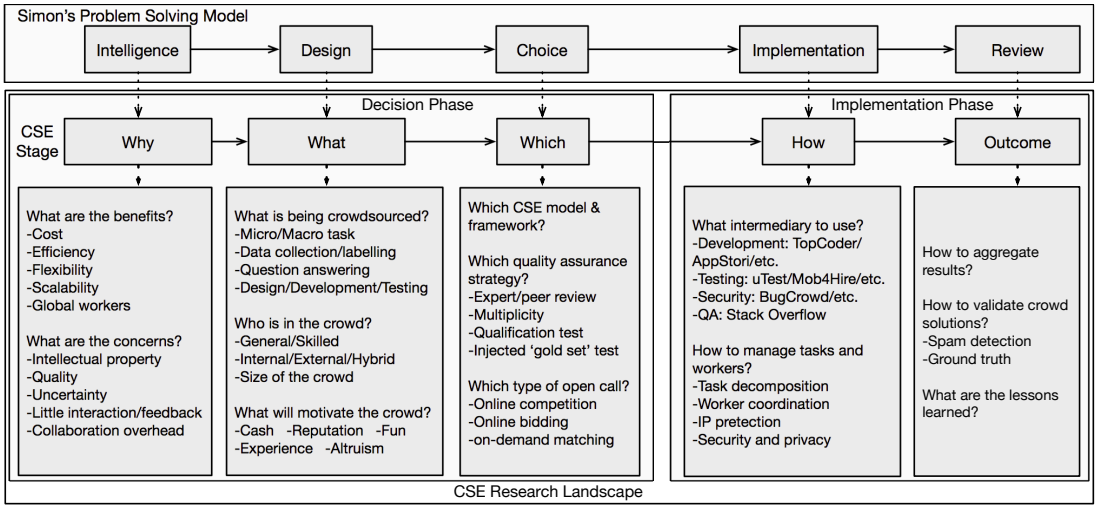



Figure 2.9: Problem-solving Model of Crowdsourced Software Engineering (adapted from the outsourcing stage model by Dibbern et al. [99])

### Commercial Platforms

A list of existing commercial crowdsourcing platforms that support software engineering are presented in Table 2.6. These platforms employ various types of open call formats, such as the widely used online competition, on-demand matching (where the workers are selected from the registrants), and online bidding (where the developers bid for tasks before starting their work). The platforms also focus on a broad range of task domains within software engineering. Platforms such as TopCoder and GetACoder support multiple types of software development tasks. Others are more specific. For example, uTest and BugCrowd are designed for software testing and security analysis, respectively. There are also general crowdsourcing marketplaces such as Amazon

Mechanical Turk (AMT), oDesk and Freelancer, which are not designed for software engineering specifically, but can, nevertheless, be used to support various software development tasks.

Different platforms may also use various process models. In the remainder of this subsection we introduce typical commercial platforms and their processes for Crowdsourced Software Engineering:

*1) TopCoder* is a pioneer for practising Crowdsourced Software Engineering. It has its unique process and development model, which is known as the TopCoder Competition Methodology. The platform supports the independent graphic design, development, data science challenges, and the development of complex software (by decomposing into multiple sub-tasks). Viewed from the top level, the systematic process may resemble the waterfall model. However, each development phase is realised through a series of online competitions in which the crowd developers compete with each other. Only qualified winning solutions are accepted. Qualified outputs are used as the inputs for the subsequent development phases. In this context, 'qualified' means passing a minimum acceptance score which is rated through a review process. The review board is also made up of crowd developers from the TopCoder community.

*2) AppStori* is a more recent platform for crowdsourcing mobile app development. It uses a crowdfunding model to fund development and attracts app developers and consumers to work closely together. The crowd developers can post their projects to raise funds from the crowd or to recruit other developers for app implementation. Consumers can propose ideas for new app development, contribute money, act as beta testers and offer feedback on existing projects. The whole development process, from conception to release, is achieved through collaboration among crowd developers and consumers.

*3) uTest* is one of the leading platforms for crowdsourced software testing. It claims to support the world's largest open community for software testing[27]. The crowd testing community enables a wide range of virtual on-demand testing services, such as functional testing, usability testing, localisation testing and load testing. The crowdsourced

[27]http://www.utest.com/about-us

testing process starts with a phase in which the clients can specify their testing needs. Flexible choices concerning testing devices, operating systems, geographic locations and budgets are provided by the platform. Appropriate testers are selected from the community, based on several metrics such as their previous performance, skills, languages, testing devices and locations. The selected testers report their testing work in real-time and submit their test report for approval. It is usually the clients' responsibility to review the submission and decide which workers are qualified to be paid for their work.

*4) StackOverflow*[28] is a question and answer website which provides crowdsourced programming knowledge for software developers. Although such crowd knowledge is passively 'pulled' by developers with issues rather than being an active part of the development process, it poses a positive impact on open source software development [371, 372] as well as conventional software development process. It has been used to improve integrated software development environments [43, 94, 319, 320, 321, 403] and software API documentation [181, 307].

*5) Bountify* is a platform similar to StackOverflow. However, it has more 'self-contained', micro programming tasks. Each yields a payment of a certain amount of money, ranging from 1 to 100 US dollars. A study on program synthesis [87] used this platform to obtain initial seeds for their genetic programming algorithm.

Other more general-purpose crowdsourcing platforms such as Amazon Mechanical Turk and CrowdFlower also have been widely used in software engineering research:

*1) Amazon Mechanical Turk (AMT)* is a popular crowdsourcing marketplace for micro-tasks. By employing crowd workers on the platform to exploit human computation, small teams may mitigate the challenges in developing complex software systems [52]. This platform has been employed to support program synthesis [87], graphical user interface (GUI) testing [101], oracle problem mitigation [308], and program verification [332] in software engineering.

*2) CrowdFlower* is a micro-task crowdsourcing platform that is similar to Amazon

[28] http://www.stackoverflow.com

Mechanical Turk. It focuses more on solving data problems such as data collection, cleaning and labelling. Afshan et al. [15] employed this platform to evaluate the human readability of test string inputs, generated by a search-based test data generation technique with a natural language model.

Several studies provided further information on existing commercial platforms for software engineering. An introduction to software crowdsourcing platforms [311] briefly summarised several platforms for collaborative software development and compared crowdsourced software development with proprietary software development, outsourced software development and open source software development. Fried et al. [121] summarised three types of crowdsourcing platforms for the software industry: platforms such as Amazon Mechanical Turk[29] that support the use of human knowledge in an inexpensive way; platforms such as TopCoder that support contest-based software development; and platforms like MathWorks[30] that support programming competitions with a unique 'competitive collaboration' feature. Wu et al. [390] proposed an evaluation framework for assessing software crowdsourcing processes with respect to multiple objectives such as cost, quality, diversity of solutions and crowd competitions. The competition relationship was evaluated by a 'min-max' (defence-offence) mechanism adapted from game theory. Based on the proposed evaluation framework, the contrast between TopCoder and AppStori software crowdsourcing processes was illustrated.

#### Case Studies

Many Crowdsourced Software Engineering case studies have been reported in recent years. Most are based on one or several commercial platforms described above. Among them, the TopCoder platform has the most case studies reported upon in the literature [33, 201, 220, 280, 281, 351, 359, 391].

Stol et al. [351] presented an in-depth case study with a client company which has crowdsourced software development experience using TopCoder. A series of issues pertaining to the TopCoder development process were identified through interviews with

[29] http://www.mturk.com
[30] http://www.mathworks.com

the client company. For instance, the platform generally followed a waterfall model, which brought coordination issues to the client company as it adopted an agile development model. Also, quality issues were pushed to later stages in the TopCoder development process, which was not regarded as best practice. The research protocol [352] contains details of the design of this case study which can be used for replicating the study. Based on the lessons learned from this case study, the authors further enunciated their advice for crowdsourced software development. For instance, the requester should provide the crowd with clear documents and avoid anonymous interaction with crowd developers [111].

Tajedin and Nevo [359] also conducted an in-depth case study in the form of interviews, but from the perspective of TopCoder's management team, rather than the client. The case study revealed two types of value-adding actions that exist in the crowdsourcing platform, i.e., the macro, market level and the micro, transaction level actions.

Wu et al. [391] highlighted the lessons learned from their collected software crowdsourcing data. Two crowdsourced software development processes employed by TopCoder and AppStori were examined. The paper argued that the 'min-max' competition behaviour contributes to the quality and creativity of crowdsourced software development.

Nag et al. [280] reported their collaboration with TopCoder to crowdsource spaceflight software development for the SPHERES Zero Program, supported by NASA, DARPA and Aurora Flight Sciences. A detailed version can be found in Nag's Master's thesis [281].

Lakhani et al. [201] described the development of TopCoder from the year of 2001 to 2009, including the evolution of the platform and the community, the benefits and concerns from the client's perspective, and the management roles and challenges of the TopCoder development process.

Archak [33] conducted an empirical analysis of developers' strategic behaviour on TopCoder. The cheap talk phenomenon [108] during the registration phase of the contest was identified, i.e., in order to soften competition, highly rated developers tend to register for the competition early thereby seeking to deter their opponents from seeking to

participate in the marketplace. Archak argued that the cheap talk phenomenon and the reputation mechanisms used by TopCoder contribute to the efficiency of simultaneous online contests. In addition, a regression analysis was performed to study the factors that affect the quality of the contest outputs. The payment and the number of requirements factors were identified as significant predictors for final submission quality. Li et al. [220] also conducted a case study on TopCoder. A set of 23 quality factors were identified from the aspects of project and platform.

Regarding the case studies that were based on the platforms other than TopCoder: Zogaj et al. [408, 409] conducted a case study on a German start-up crowd testing platform called *testCloud*. Three types of challenges were highlighted in the case study: managing the crowd, managing the process and managing the techniques. Bergvall-Kareborn and Howcroft [54] reviewed Apple's business model for crowdsourcing mobile apps. By reporting fieldwork among Apple mobile app developers in three countries, they showed how the company benefited from crowdsourcing, e.g., effectively outsourced their development tasks to global online developers while sidestepping some costs incurred by directly employing high-tech workers.

Some case studies focused on region-specific practices in crowdsourced software development. For example, one case study [238, 324] presented the preliminary results of a multi-year study on crowdsourcing in the Brazilian IT industry. This study reported interviews that highlighted the generally low awareness of software crowdsourcing and concerns about the crowdsourced software quality. Phair's doctoral thesis [315] reported a qualitative case study on using crowdsourced software development to implement a web application for a non-profit organisation. Benefits such as measurable cost savings and an increased ability to work on multiple projects were identified. A few other case studies have reported the practice of software crowdsourcing in specific domains, such as crowdsourced proteomics software development [253] and crowdsourced e-government software development [337, 379].

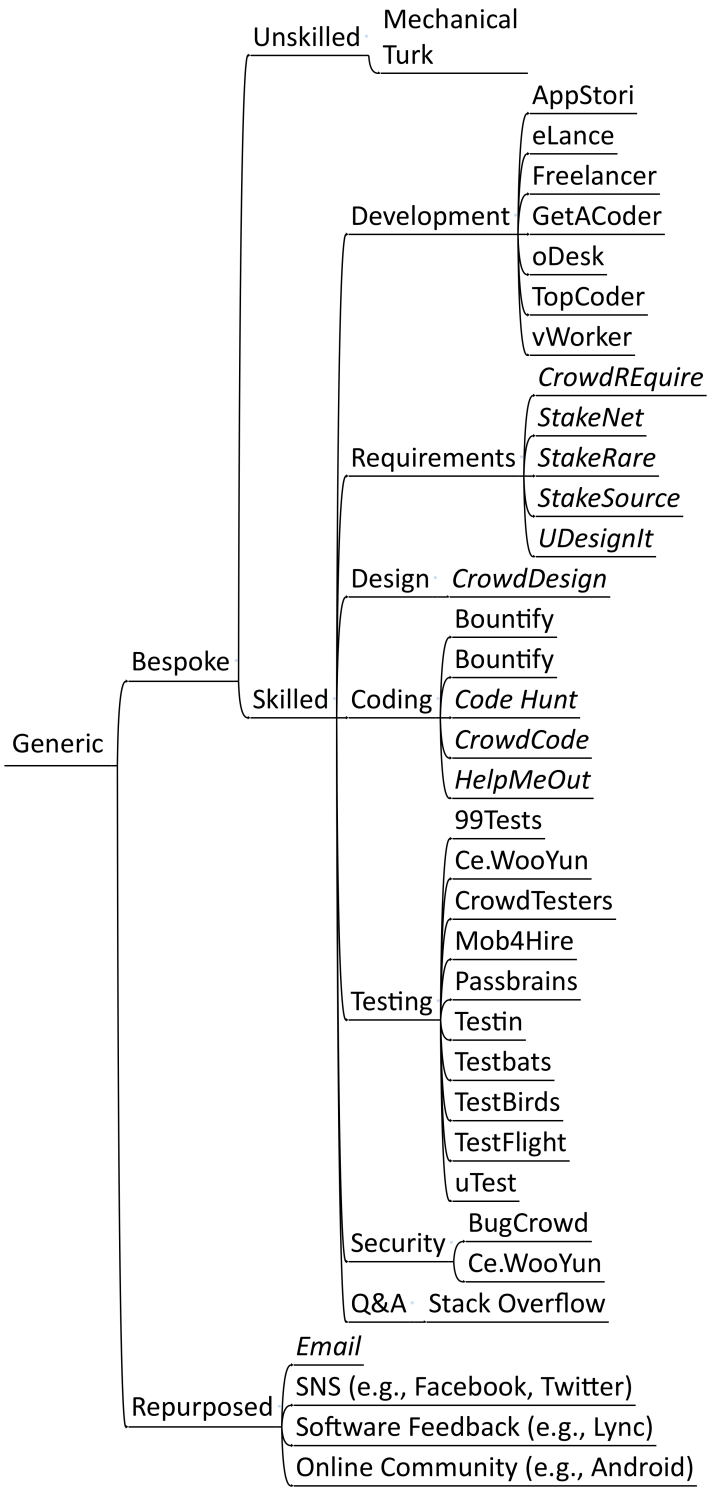



Figure 2.10: Scheme of crowdsourced software engineering platforms (The *italic text* indicates an experimental/non-commercial platform.)

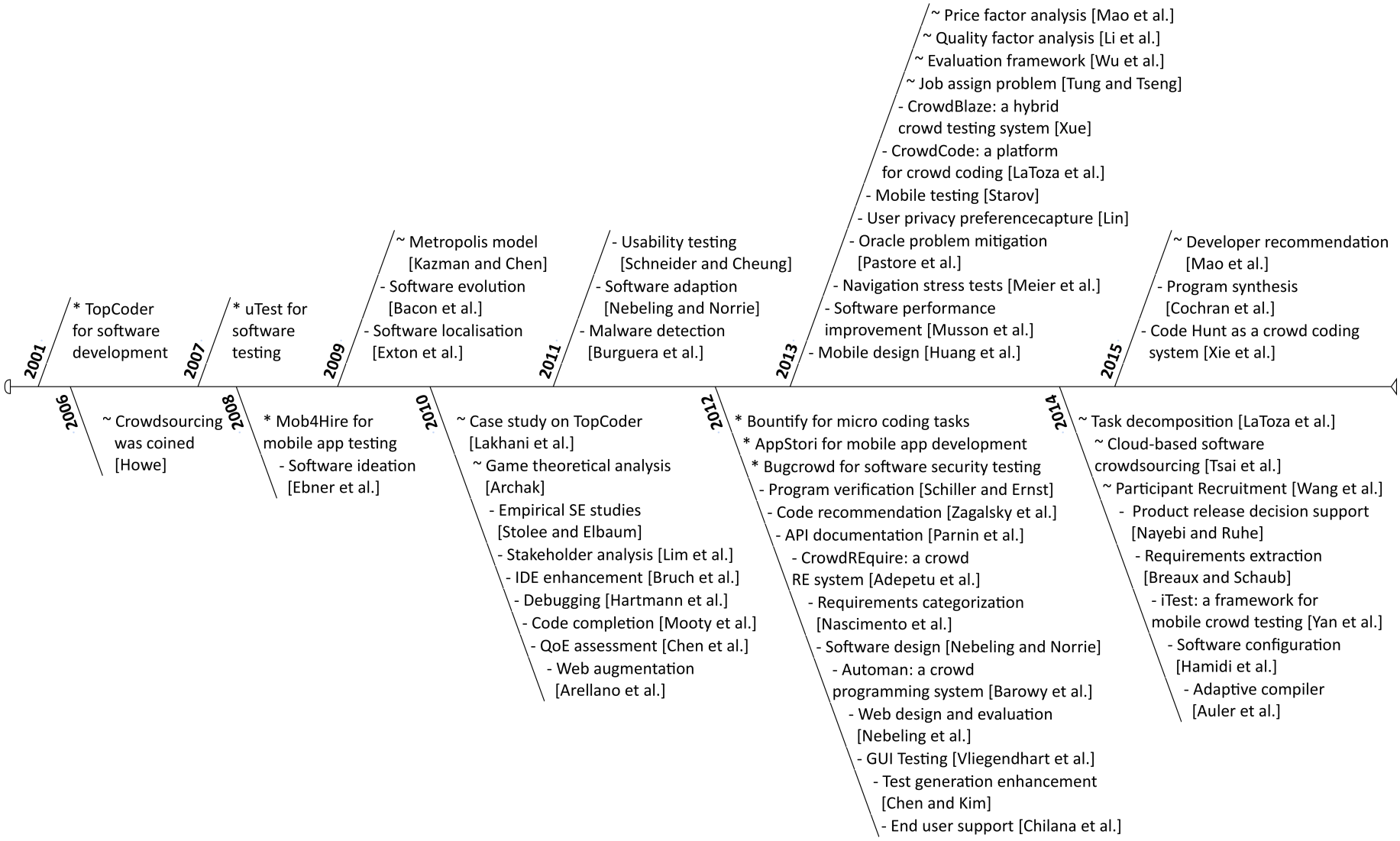



Figure 2.11: Timeline for the development of Crowdsourced Software Engineering ('*' indicates the establishment of a platform. '~' shows the first practical/theoretical study and '-' stands for the first application work)

### 2.2.4 Crowdsourcing in Software Testing and Verification

Software testing and verification have received considerable attention in the software engineering research community. It is therefore unsurprising that we found the number of related crowdsourcing studies dominate those of other categories.

**Crowdsourcing for Software Testing** Crowdsourcing for software testing is often termed 'Crowdsourced Testing' or 'Crowd Testing'. Compared with traditional software testing, crowdsourced software testing has the advantage of recruiting, not only professional testers, but also end users to support the testing tasks.

Crowdsourcing has been applied to various types of testing activities, including usability testing [132, 235, 258, 287, 288, 334, 361], performance testing [278], GUI testing [101, 375], test case generation [78, 317], and the oracle problem [308]. We discuss each of these below:

*1) Usability Testing:* Traditional usability testing is labour-intensive and can be expensive and time-consuming [235]. Recruiting online ad-hoc crowd labour may be a way to ameliorate these issues, by exploiting a large potential user pool and providing lower labour rates with extended incentives to the end users. Crowdsourced usability testing has demonstrated its capability for detecting usability problems as good as the testing done by 'experts' [334]. However, Liu et al. [235] showed that the quality of crowdsourced usability testing was worse than that of the face-to-face usability testing in a laboratory setting. Nebeling et al. [288] further discussed this issue and suggested that the advantages outweigh disadvantages according to their results. Nevertheless, these existing studies agree on the benefits of cost saving, fast delivery as well as easy access of crowdsourced usability testing.

Schneider and Cheung [334] first demonstrated the viability of employing on-demand crowd users for usability testing. They also proposed methods to help observe the testers during the process. Liu et al. [235] conducted a comparative study on crowdsourced and traditional laboratory usability testing. Their experimental results highlighted quality issues and the challenge of detecting 'cheating behaviour'. Nebeling et al. [287, 288] proposed a framework with a toolkit implementation named *CrowdStudy*

for crowdsourced website usability testing. For identifying outliers in the crowdsourced usability testing results, Gomide et al. [132] proposed an approach that employs deterministic automata for automatic hesitation detection. The idea is to capture users' biofeedback from mouse movements and a skin sensor, for revealing their hesitation behaviours. This can be useful in filtering non-confirming usability testing results.

*2) Performance Testing:* Software performance in a real-world setting can be hard to test due to the various user behaviours and execution environments. Musson et al. [278] proposed an approach, in which the crowd was used to measure real-world performance of software products. The work was presented with a case study of the Lync[31] communication tool at Microsoft. The study indicated the usefulness of the approach for identifying performance issues and assisting development team with decision making. In this case, the Lync software itself is repurposed as the crowdsourcing platform, and there is an implicit open call (i.e., permission grant request) for providing performance data from the crowd users. Other similar cases for such crowdsourced performance testing include the Chrome's[32] and Firefox's[33] built-in telemetries (performance testing frameworks) [18].

*3) GUI Testing:* Automated GUI test case generation is difficult, while manual GUI testing is too slow for many applications [259]. It is a challenging task to test a GUI continuously. Crowdsourcing is considered as a promising approach for continuous GUI testing [101].

Vliegendhart et al. [375] first proposed GUI testing for multimedia applications. Crowd testers were recruited from Amazon Mechanical Turk. They were asked to carry out A/B tests of user interfaces via remote virtual machines. Their experimental results indicated that it took less than three days and 50 US dollars to complete two featured GUI testing tasks with 100 assignments each. Based on this crowd performance, it was concluded that user connection speed was not an issue in their study. However, the quality of the testing results was not reported in this study.

Dolstra et al. [101] also demonstrated the possibility of crowdsourcing GUI tests by

[31] http://office.microsoft.com/lync
[32] http://www.chromium.org/developers/telemetry
[33] http://telemetry.mozilla.org

offering remote virtual machines to testers, recruited from Amazon Mechanical Turk. The experimental results showed feasibility and reliability of the proposed approach.

*4) Test Case Generation:* Test cases are essential to ensure software quality. Although a number of automatic test case generation methods have been proposed, their test coverage is not ideal [203], due to several non-trivial tasks that are difficult for programs but may not be so hard for humans [78]. Chen and Kim [78] investigated object mutation and constraint solving issues, underlying existing test generation tools such as jCUTE [335], Randoop [303] and Pex [363]. A Puzzle-based Automatic Testing (PAT) environment was presented for decomposing and translating the object mutation and constraint solving problems into human-solvable games (gamification). Experimental results from two open source projects showed 7.0% and 5.8% coverage improvement, compared to the coverage of two state-of-the-art test case generation methods.

Pham et al. [316] conducted a study on the testing culture of the social coding site — GitHub, and found that capable developers sometimes solve issues in others' repositories in a fast and easy manner, which is called the *drive-by commit* phenomenon. This phenomenon has the potential to be leveraged for generating test cases in social coding sites [317]. However, it is still a conceptual idea which remains to be realised in future work.

*5) Oracle Problem:* An oracle is typically needed to determine the required output of a program for a given input [49, 384]. Such oracles may need to rely on human input [312], which makes it hard to fully automate software testing. Pastore et al. [308] investigated crowdsourcing to mitigate the oracle problem. They crowdsourced automatically generated test assertions to a qualified group of workers (with programming skills) and an unqualified group of workers on Amazon Mechanical Turk. Workers were asked to judge the correctness of the assertions and further fix false assertions. The experimental results suggested that crowdsourcing can be a viable way to mitigate the oracle problem, although the approach requires skilled workers provided with well-designed and documented tasks.

To support the application of crowdsourcing for software testing, especially for mobile application testing, several frameworks have been proposed [223, 397, 398]:

*CrowdBlaze* [397] is a crowd mobile application testing system which combines automatic testing and human-directed interactive testing. This study aimed to use redundant resources to help improve software systems. *CrowdBlaze* initially explores the app with static analysis and automatic testing, and then recruits crowd users to provide input for complex cases which enable automatic testing to further explore the app. Compared to employing automatic testing alone, the proposed system was demonstrated to cover 66.6% more user interfaces according to the evaluation results.

*iTest* [398] is a framework for mobile apps with more automation features than existing industrial mobile application testing service platforms such as uTest and Mob4Hire: the crowd testers are selected via a greedy algorithm, and the generated test results and logs in the framework are submitted automatically.

*Caiipa* [223] is a cloud service for scalable mobile application testing. The service framework is equipped with a unique contextual fuzzing approach to extend the mobile app running context space. It uses both crowdsourced human inputs and crowdsourced measurements, such as various network conditions, with multiple operator networks and different geographic locations. Experimental results suggested that *Caiipa* has the capability to uncover more bugs compared to existing tools with none or partial mobile contexts.

Xie [393] summarised three types of cooperative testing and analysis: human-tool, tool-tool and human-human cooperation. The crowd-supported software testing and analysis falls into the human-human type of cooperation according to this study.

Besides, crowdsourcing has also been applied to general software evaluation [57, 341, 342] and more specific evaluation of Quality of Experience (QoE) [77, 125, 159, 160].

**Crowdsourcing for Software Verification**

Current software verification techniques generally require skilled workers, thereby raising cost issues. Crowdsourcing may reduce the skill barriers and costs for software verification [19, 100, 221, 332, 333].

DARPA published a solicitation for game-based large scale software verification in 2011, which is named the Crowd Sourced Formal Verification (CSFV) program. A series of

research and practice [100, 104, 382] were conducted under this program. Dietl et al. [100] proposed to use gamification to attract a general crowd as a verification workforce. The 'verification games' approach transforms a verification task into a visual game that can be solved by people without software engineering knowledge.

Li et al. [221] presented a system called *CrowdMine* for recruiting non-expert humans to assist with the verification process. The system represents simulation or execution traces as images and asks the crowd of humans to find patterns that fail to match any pre-defined templates.

Schiller and Ernst [332] developed a web-based IDE called *VeriWeb* for reducing the barriers to verified specification writing. The IDE was designed to break down a verified specification writing task into manageable sub-problems. The experimental results suggested time and cost benefits. However, the workforce needs to be contracted workers rather than ad-hoc labours provided by crowdsourcing markets such as Amazon Mechanical Turk. A more detailed version of this study can be found in Schiller's doctoral thesis [333].

### 2.2.5 Crowdsourcing in other Software Engineering Activities

Crowdsourcing applications to software engineering are presented as multiple subsections, according to the software development life-cycle activities that pertain to them. The following major stages are addressed: software requirements, software design, software coding, software testing and verification, software evolution and maintenance. An overview of the research on Crowdsourced Software Engineering is shown in Table 2.7. The references that map to each of the software engineering tasks are given in Table 2.8. The commercial and experimental crowdsourcing platforms in these studies follow the scheme in Figure 2.10.

A timeline of the introduction of various ideas and concepts is illustrated in Figure 2.11. For example, starting from 2009, crowdsourcing was employed to help evolve software and its localisation. Most recently, the crowdsourcing model was used for program synthesis. Other important events and theoretical/practical studies that can reflect the

development of Crowdsourced Software Engineering are also illustrated in the timeline.

For the Crowdsourced Software Engineering studies with empirical evaluations, we summarised the conducted experiments in Table 2.9, to reveal the detailed experimental settings and results. With the summary, we calculated the distributions of the crowd size, cost and the platforms used in Crowdsourced Software Engineering experiments, as shown in Figure 2.12 and Figure 2.13 respectively.

### Crowdsourcing for Software Requirements Analysis

Requirements analysis is a widely accepted critical step that impacts the success of software projects [348]. A series of studies [14, 20, 63, 134, 157, 158, 224, 226, 227, 228, 229, 275, 282, 283, 336, 346, 377] have investigated crowdsourcing to support this process.

Traditional stakeholder analysis tools require experts' manual effort to extract stakeholders' information. Lim et al. [228] proposed *StakeSource* to identify crowdsourced stakeholders involved in a stakeholder analysis process. This tool was designed to reduce the cost of reliance on experts to approach stakeholders. It was a complementary to their previously proposed *StakeNet* [224], which recommends stakeholders via social networking. The authors further improved this tool and proposed *StakeSource2.0* [229]. The new version integrates support for identifying stakeholders and prioritising their requirements. *StakeSource2.0* was used to automate the stakeholder identification and prioritisation step of the *StakeRare* [226] method, an approach for large-scale requirements elicitation based on social network analysis and collaborative filtering techniques. Lim and Ncube [227] subsequently showed the application of the tool for system of systems projects. The tool is publicly available online[34].

Hosseini et al. [157] focused on employing crowdsourcing for requirements elicitation. They summarised the main features of the crowd and crowdsourcer in crowdsourced requirements engineering by reviewing existing literature. A preliminary result of a survey conducted on two focus groups was reported to reveal the relationship between

[34]http://www.cs.ucl.ac.uk/research/StakeSource

Table 2.7: An overview of the research on Crowdsourced Software Engineering

| SE Phase | SE Task | Why | Bespoke Tool | Stakeholder | | |
|---|---|---|---|---|---|---|
| | | | | Requester | Platform | Worker |
| Requirements | Requirements Acquisition | Cost, User needs, Domain knowledge, Automation, Quality | StakeSource, StakeSource2.0, StakeNet, StakeRare, iRequire | Requirements engineers, Designers, Software teams, Researchers | Email, StakeSource, StakeSource2.0, StakeNet, StakeRare, CrowdREquire, UDesignIt, Bespoke, AOI, AMT | All stakeholders, Users, Undefined crowd |
| | Requirements Categorisation | User needs | None | Requirements engineers, Designers | Unspecified | Users |
| Design | UI Design | User needs, Quality, Diversity | None | Designers, Non-technical end users | Bespoke, AMT, CrowdDesign, Email | Users |
| | Architecture Design | Quality, Diversity | None | Researchers | Email | Designers |
| | Design Revision | Quality, Diversity | None | Researchers | Email | Designers |
| Coding | IDE Enhancement | Debugging, API aid | BlueFix, Calcite, Example Overflow, Seahawk, Prompter, SnipMatch | Developers | HelpMeOut, Stack Overflow, oDesk | Developers |
| | Program Optimisation | Human solutions | None | Developers, Researchers | Bountify, AMT, Software Feedback | Developers, Undefined crowd, Users |
| | Crowd Programming Support | Automation, Human solutions | CrowdLang, CIDRE, Collabode | Developers, Teachers | Bespoke, AMT, Code Hunt | Users, Developers |
| Testing | Usability Testing | Cost, Time | CrowdStudy | Testers | CrowdStudy, Bespoke, AMT, CrowdFlower | Users |
| | Performance Testing | Real-world measure | None | Client companies | Lync | Users |
| | GUI Testing | Cost, Scalability | None | Testers | AMT | Undefined crowd |
| | QoE Testing | Cost, Diversity | Quadrant of Euphoria | Researchers | Quadrant of Euphoria, Bespoke, AMT, Microworkers | Undefined crowd |
| | Test Generation | Human inputs | PAT | Testers, Researchers | Twitter | Undefined crowd |
| | Oracle Problem | Human solutions, Automation | None | Testers, Researchers | AMT | Qualified / Unqualified crowd |
| | Crowd Testing Support | Human inputs | CrowdBlaze | Testers, Researchers | Bespoke, AMT, Mobileworks, Email | Undefined crowd |
| | General Evaluation | User needs, Diversity | None | Researchers | Bespoke, AMT | Users |
| Verification | Non-expert Verification | Cost, Speed | Verification Games, VeriWeb | Developers, Researchers | Bespoke, AMT, vWorker | Undefined crowd |
| Evolution | Software Adaptation | User needs, Cost, Diversity, Speed | MoWA, CrowdAdapt | Developers, Designers, Users, Researcher | Bespoke, Facebook, Online community | Users |
| Maintenance | Software Documentation | Domain knowledge | COFAQ | Developers, Researchers | Q&A, Stack Overflow, SciPy Community | Developers, Researchers |
| | Software Localisation | Domain knowledge, Cost, Speed | None | Developers, Researchers | AMT | Undefined crowd |
| Other | Security and Privacy Augmentation | Diversity, Domain knowledge, User needs | Crowdroid, Modding-Interface, CrowdSource, SmartNotes, ProtectMyPrivacy | Developers, Researchers | Android User Community | Users |
| | End User Support | Domain knowledge | LemonAid | Developers, Researchers | AMT | Users |
| | Software Ideation | User needs, Open innovation, Recruitment | SAPiens, IdeaMax | Client Companies | Repurposed, Bespoke | Users |

Table 2.8: Reference mapping of the research on Crowdsourced Software Engineering

| SE Phase | SE Task | Reference |
|---|---|---|
| Requirements | Requirements Acquisition | [14, 20, 63, 134, 157, 158, 224, 226, 227, 228, 229, 275, 282, 283, 336, 346, 377] |
| | Requirements Categorisation | [275, 282] |
| Design | UI Design | [55, 206, 212, 286] |
| | Architecture Design | [212] |
| | Design Revision | [212] |
| Coding | IDE Enhancement | [43, 50, 67, 68, 76, 94, 109, 154, 187, 272, 318, 319, 320, 321, 322, 383, 386, 403] |
| | Program Optimisation | [40, 87] |
| | Crowd Programming Support | [48, 128, 129, 130, 263, 264, 394] |
| Testing | Usability Testing | [132, 235, 258, 287, 288, 334, 361] |
| | Performance Testing | [278] |
| | GUI Testing | [101, 375] |
| | QoE Testing | [77, 125, 159, 160] |
| | Test Generation | [78, 317] |
| | Oracle Problem Mitigation | [308] |
| | Crowd Testing Support | [223, 397, 398] |
| | General Evaluation | [57, 341, 342] |
| Verification | Non-expert Verification | [100, 221, 332, 333] |
| Evolution | Software Adaptation | [19, 20, 21, 24, 45, 73, 140, 155, 237, 284, 285, 289] |
| Maintenance | Software Documentation | [50, 75, 181, 307, 309] |
| | Software Localisation | [107, 135, 244, 262] |
| Other | Security and Privacy Augmentation | [17, 36, 69, 169, 230, 231, 306, 330, 339] |
| | End User Support | [80, 81, 82] |
| | Software Ideation | [102, 175, 176, 200] |

these features and the quality of the elicited requirements. Wang et al. [377] also used crowdsourcing to acquire requirements, but with a focus on overcoming the problem of recruiting stakeholders with specific domain knowledge. They proposed a participant recruitment framework, based on spatio-temporal availability. Their theoretical analysis and simulation experiments demonstrated the feasibility of the proposed framework.

The crowd stakeholders are not only a source of requirements, but also can help with requirements prioritisation and release planning. Nascimento et al. [282] investigated the use of crowdsourcing for requirements categorisation based on Kano's model. The model uses a questionnaire to help classify requirements into five categories. The value of each requirement for a given user is identified in their approach. A framework was proposed for finding stakeholders involved in the process. Nayebi and Ruhe [283] presented the Analytical Open Innovation (AOI) approach to assist developers in making release decisions. The crowdsourcing model enables the AOI approach to systematically gather information from customers and other stakeholders. An illustrative case study was presented as a proof-of-concept to demonstrate the key ideas of the AOI approach.

Non-professional crowd workers have been used to process requirements documents. This is a laborious task when performed manually, to extract requirements from large natural language text source. However, such data are frequently needed as the ground truth for evaluation. This limits the generalisation of evaluations to automatic requirements extraction methods. Breaux and Schaub [63] conducted three experiments concerned with employing untrained crowd workers to manually extract requirements from privacy policy documents. Experimental results indicated a 16% increase in coverage and a 60% decrease in cost of manual requirements extraction, with the help of their task decomposition workflow.

To support crowdsourced requirements engineering activities, Adepetu et al. [14] proposed a conceptualised crowdsourcing platform named *CrowdREquire*. The platform employs a contest model to let the crowd compete with each other to submit requirements specification solutions to the client defined tasks. The business model, market strategy and potential challenges such as quality assurance and intellectual property issues of the platform were also discussed.

Table 2.9: An overview of the crowdsourcing experiments conducted in the application papers

| Phase | SE Task | Platform | Crowd | Size | Subject | Effort | Reward | Result | Reference |
|---|---|---|---|---|---|---|---|---|---|
| Requirements | Requirements Elicitation | StakeNet | Stakeholders | 68 | RALIC[a] | - | - | StakeNet can identify stakeholders and their roles with high recall, and can prioritise them accurately. | [224] |
| | Requirements Elicitation | StakeRare | Stakeholders | 87 | RALIC | - | - | StakeRare can predict and prioritise stakeholder needs accurately. | [226] |
| | Requirements Extraction | AMT | Unskilled | 76<br>38<br>- | -<br>-<br>- | 448 classifications<br>135 classifications<br>- | $0.15 per task<br>$0.15 per task<br>$0.08-$0.10 per task | The approach can reduce 60% cost and increase 16% coverage in manual extraction. | [63] |
| Design | Architecture Design | Email | Students | 20 | a traffic flow simulation program | 12.9 hours (average) | $100 each person + 4*$1000 prizes | All participants borrowed others' design ideas and most improved the design quality. | [212] |
| | User Experience Design | | | 20 | | 21.3 hours (average) | | | |
| Coding | IDE Enhancement | StackOverflow | StackOverflow community | - | 12 Java, 12 C++ and 11 .NET programming tasks | - | - | For 77.14% of the assessed tasks at least one useful Q&A pair can be recommended. | [94] |
| | IDE Enhancement - Code Annotation | oDesk | Developers | - | 500 Ruby code snippets | 500 annotations | - | Among the annotated snippets, 86% correspond to a useful programming task; 96% can be embedded into a standalone function, and 91% do not have another more common form. | [109] |
| | Program Synthesis | Bountify | Developers | 5 | 4 regular expression tasks | wrote 14 regular expression | $10 | Consistent prgoram boosts in accuracy can be achieved with modest monetary cost | [87] |
| | | AMT | Unskilled | 5 | | classified strings as valid or invalid | $0.05-$0.25 per task | | |
| Testing | System evaluation | AMT | Unskilled | 65<br>69 | Web queries from semantic search engines | 579 HITs[b]<br>421 HITs | $347.16 ($0.20 per HIT)<br>- | Crowdsourced evaluation tasks can be repeated over time and maintain reliable results. | [57] |
| | Usability Testing | AMT<br>AMT+CF[c] | Unskilled | 11<br>44 | a graduate school website | 11 HITs<br>44 HITs | $2.92 ($0.15 per HIT)<br>$347.16 ($0.20 per HIT) | Reduced cost and time. However quality was worse compared to the testing in a lab setting. | [235] |
| | Usability Testing | AMT | Unskilled | 28 | a subpage of a university website | 4-5m (average) to answer all 8 questions | - | The crowdsourced usability testing shared similar results with a laboratory setting. | [258] |
| | Usability Testing | CrowdStudy + AMT | Unskilled | 93 | A news article page | 28 custom layouts, 143 ratings and 32 answers | - | The usefulness and the ability to be configured for different scenarios were demonstrated. | [288] |
| | | | | 84 | Wikipedia website | 33 different type tasks | - | | |
| | Performance Testing | Lync | End users | 48,000 | Lync (Microsoft instant-messaging client) | Usage behaviours | 0 | The approach had been successfully deployed and had improved development decisions at Microsoft. | [278] |
| | GUI Testing | AMT | Unskilled | 100<br>100 | Tribler | 100 assignments<br>100 assignments | $25<br>$25 | The approach was able to evaluate an experimental UI feature within a few days at low costs. | [375] |
| | GUI Testing | AMT | Unskilled | 398 | Tribler, KDE, Xfce[d] | 700 assignments | $0.10-$0.15 per HIT | The approach is feasible and reliable, although there was a quality issue in continuous testing. | [101] |
| | QoE Testing | Quadrant of Euphoria + AMT | Unskilled | - | a three-minute-long speech audio file and a benchmark video clip | 2130 runs of experiments | $21.3 | With consistency assurance crowdsourcing can yield results as well as laboratory experiments. | [77] |
| | QoE Testing | Microworkers | Unskilled | 10,737<br>1,593 | 3 videos with different content classes | 10,737 ratings<br>1593 ratings | $0.2625 per rating<br>$0.0834 per rating | The proposed methods represented a step in making crowd-testing sufficiently mature for wide adoption. | [125] |
| | Test Generation | Twitter | Unskilled | 120 | the Apache Commons Math and Collections libraries | 1 minute per puzzle | 0 | 84 of the top 100 constraint solving puzzles and 24 of the top 100 object mutation puzzles were successfully solved. | [78] |
| | Oracle Problem Mitigation | AMT<br>AMT | Unqualified<br>Qualified | -<br>- | the Java Stack class<br>the Java Stack class, the Java libraries Trove4J and j8583, | 200 assignments<br>500 assignments | $0.15-$0.20 per assignment | CrowdOracles is a promising solution to mitigate the oracle problem, however geting useful results from an untrained crowd is difficult. | [308] |
| | Crowd Testing Support | Mobileworks + Email | End users | 75 | 8 popular Android apps from Google Play | - | - | Statically Aided Interactive Dynamic Analysis consistently obtained greater coverage than other techniques. | [397] |
| Verification | Non-expert Verification | vWorker<br>AMT | Developers<br>Unskilled | 14<br>< 10 | StackAr (an array-based stack in Java) | 3 hours (average)<br>- | $6-$22 per hour<br>> $0.25 per HIT | VeriWeb can save time and money with contracted workers. Current ad-hoc labours are not well-suited for verification. | [332] |

[a] RALIC is the acronym of 'Replacement Access, Library and ID Card', which is an access control system developed by University College London.

[b] A Human Intelligent Task (HIT) is a single, self-contained task that workers can perform.

[c] AMT and CF refer to Amazon Mechanical Turk and CrowdFlower, respectively.

[d] Tribler is a peer-to-peer file-sharing client. KDE and Xfce are two desktop environments for Unix-like platforms.

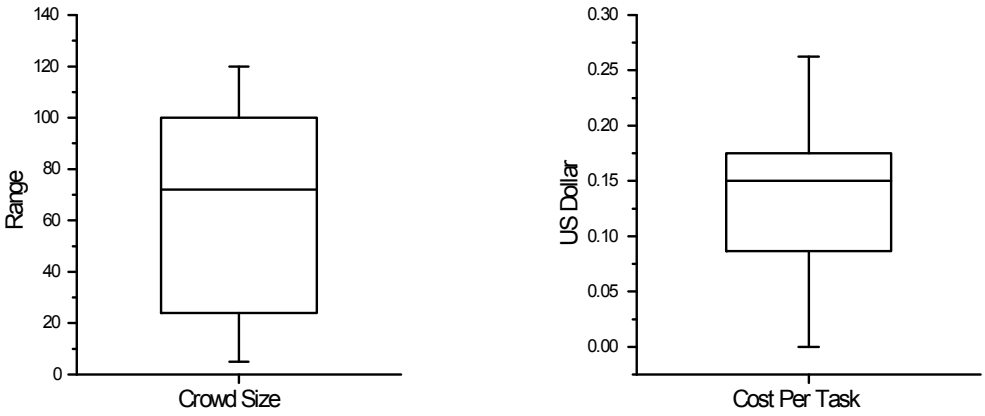



Figure 2.12: Crowd size and cost per task in the surveyed studies

**Crowdsourcing for Software Design**

Among existing commercial crowdsourcing marketplaces, there are many platforms supporting software interface design, such as 99designs, DesignCrowd and crowdSPRING. However, few research studies have been reported on the performance of using crowdsourcing for software design.

In order to provide software designers inspiring examples during the wireframing stage, Huang et al. [164] leveraged the crowd to map between mobile app wireframes and design examples over the Internet. Lasecki et al. [206] proposed a crowdsourcing system named *Apparition* to help designers prototype interactive systems in real-time based on sketching and function description. Experimental results showed that *Apparition* was able to achieve an accuracy higher than 90% regarding user's intent, and to respond in only a few seconds.

Fewer crowdsourcing platforms support software architecture design. TopCoder is one of the widely used platforms. However, industrial crowdsourcing platforms such as TopCoder have limitations in evolving designs from multiple designers' solutions [212]. LaToza et al. [212] let designers produce initial designs and evolve their solutions based on others' solutions. Their study demonstrated the usefulness of recombination in crowdsourced software designs. A few suggestions on improving software design competitions were also highlighted based on their findings.

Nebeling et al. [286] also proposed to evolve software designs based data and functional-

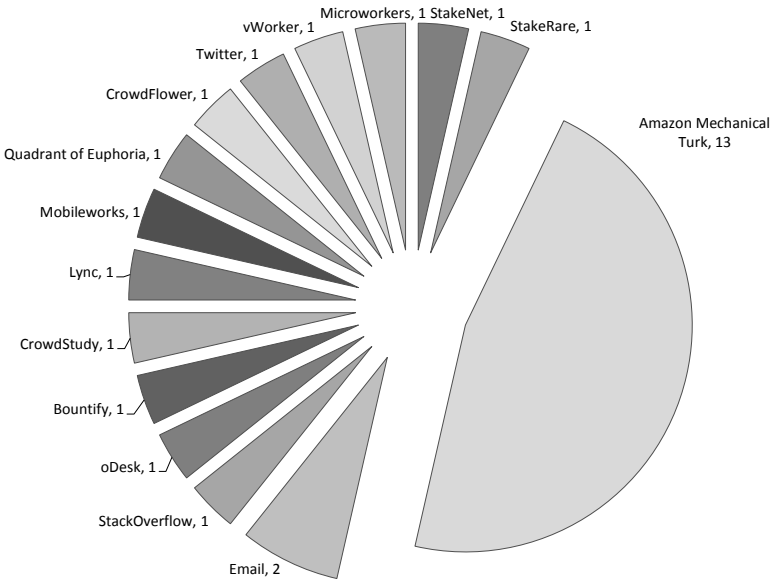



Figure 2.13: Platforms used in the surveyed studies

ity contributed by the crowd. However, the designs are specifically website components within the web engineering domain. Two preliminary experiments were conducted to show the capability of the proposed approach. Crowd motivation, quality assurance, security and intellectual property issues were also briefly discussed.

### Crowdsourcing for Software Coding

Using crowdsourcing for software coding has focused on three sub-areas: crowd programming environments, program optimisation and integrated development environment (IDE) enhancement. Unlike work that crowdsources software engineering tasks directly to the general public, studies on crowd programming environments and IDE enhancement indirectly use crowdsourcing to support software engineering activities: The studies in the former category enable the use of crowdsourcing for coding tasks in building software. Those in the latter category tend to leverage existing pre-collected crowd knowledge to aid software coding and/or debugging. We also include these papers in this survey, because they meet our inclusion criteria on using crowdsourcing in software engineering.

*1) Crowd programming environments:* Crowdsourcing intermediaries play a key role in managing and coordinating the crowd workers to accomplish the requesters' tasks.

Many studies focused on providing systems to support crowd-based coding tasks [48, 128, 129, 130, 210, 263, 264, 394].

Goldman [128] proposed role-specific interfaces for coordinating collaborative crowd coding work. By building *Collabode*, a real-time web-based IDE, the author aimed to enable emerging highly-collaborative programming models such as crowd programming. Ball et al. [48] demonstrated the design of the Cloud-based Integrated Development and Runtime Environment (CIDRE), and its implementation *TouchDevelop* [364]. CIDRE consists of three components: a crowd developer community, an online IDE and an app store. These components link the IDE designers, application developers and users together and promote the mutual feedback among them during the development process.

Xie et al. [394] proposed to use *Code Hunt* [56] from Microsoft Research as a platform for crowd programming. The platform provides coding duel games with various difficulty levels to attract online developers' participation. By carefully designing the coding duels, the platform can serve for software construction purpose by leveraging the best solutions from the crowd. Also, by recording the crowd developers' duel solving process, the multiple attempts with evolving code versions can serve for education purpose.

*2) Program optimisation:* More recently, crowdsourcing has been used to support compilation optimisation [40] and program synthesis [87].

Auler et al. [40] presented a crowdsourced adaptive compiler for JavaScript code optimisation. A compiler flag recommendation system was built in the cloud, based on the application performance data gathered from web clients. The system was used to guide the compiler to perform optimisation for a certain platform. Experiments were conducted on three optimisation implementations by JavaScript code emission for eight platforms. One of the best optimisation performance showed an average of five-fold increase in execution speed.

Cochran et al. [87] proposed an approach called *Program Boosting*, which uses crowd knowledge and genetic programming to help tackle hard programming tasks such as writing robust regular expressions for URLs. *Program Boosting* relies on two different

types of crowds for 'boosting' a program: one 'expert' crowd for generating initial candidate programs and the other 'user' crowd for evaluating the outputs (e.g., the validity of URLs) generated from the programs being evolved. The solutions obtained from the expert are used as the first population, which is subsequently evolved (by genetic programming) to yield improved solutions. Evaluation from the user crowd contributes to the evolution process. Experimental evaluation was performed on four regular expression writing tasks (to represent URLs, emails, phone numbers and dates). Experimental results showed that an average improvement of 16.25% in accuracy could be achieved on the initial human solutions.

*3) IDE enhancement:* Using crowd knowledge to support coding activities in integrated development environments has been extensively studied since 2010. Several tools and methods have been proposed to help the developers with coding and debugging [43, 50, 67, 68, 76, 94, 109, 154, 187, 272, 318, 319, 320, 321, 322, 383, 386, 403], each of which we describe below:

*HelpMeOut* [154] is a social recommender system that assists debugging with crowd-sourced suggestions. The system has a database that stores fixes for coding errors constructed by crowd developers. For collecting the fixes, the system automatically tracks code changes over time and records actions that make the error code become error-free. The evaluation was performed with novice developers through two three-hour workshops. The results showed the proposed approach was able to recommend useful fixes for 47% of the errors. However, *HelpMeOut* only supports static, compiled programming languages such as Java. To further support dynamic, interpreted web programming languages, another tool named *Crowd::Debug* [276] was proposed. More recently, Chen and Kim [76] presented the 'crowd debugging' technique that reveals defective code blocks and further suggests solutions for fixing the defects. To achieve the automated detection and recommendation, mass crowd knowledge (question-answer pairs from Stack Overflow) were collected and analysed. The proposed approach was able to identify 171 bugs in 8 high-quality open source Java projects.

The idea that a crowd of developers may be able to provide recommendations of patches for software systems finds a strong resonance in recent work on genetic improvement

[204, 300, 314, 385], and in particular work on automated bug fixing (aka 'patching' or 'automated program repair') [213]. Genetic improvement seeks to automatically improve software systems by suggesting modifications that improve functional and non-functional properties. Genetic improvement regards program code as genetic material to be manipulated in the automated search for improvements. Recent results have demonstrated the potential for this technique improve real work program's speed [204, 205, 300, 314, 385], energy [66, 219, 241] and dynamic memory [389] consumption and functionality, both by fixing bugs [214] and by adding new features [150]. Work on automated repair has also harvested human developed patches in order to improve the automated reparation process [291]. It therefore seems reasonable to conclude that hybridised versions of automated repair and social recommender systems (like *HelpMeOut*) could be extremely successful, a topic to which we return in Section 2.2.7.

*BlueFix* [383] is an online tool concerned with the problem of interpreting and understanding compiler error messages for novice programmers. An evaluation was performed based on an audience of 11 novice student programmers. The results indicated that the tool was able to help the students fix compile-time errors faster, and when compared with *HelpMeOut*, *BlueFix*'s suggestions were 19.52% higher in precision.

*Calcite* [272] is an Eclipse plugin that specifically focuses on constructor API comprehension and correct usage. The plugin uses a database that contains common object construction examples by collecting code from the web. According to a reported user study, this plugin can help developers to increase their completion rate by 40%.

*Example Overflow* [50, 403] is a code search system which utilises crowd knowledge from question and answer (Q&A) websites for suggesting embeddable code with high quality. The code snippets were collected from Stack Overflow via its public API. The search function is based on Apache Lucene. A preliminary evaluation on a subset of coding tasks indicated that the results suggested by the system were better than the ones from other existing tools studied in the experiments.

*Seahawk* [43, 319, 320, 321] is an Eclipse plugin, the aim of which has some resonance with *Example Overflow*. It seeks to utilise crowd knowledge in Q&A websites such as *StackOverflow* for documentation and programming support. Compared to *Example*

*Overflow*, *Seahawk* integrated Q&A services into IDEs and provided more friendly UI features. For example, it was found to be better at formulating queries automatically, based on code entities and providing interactive search results. It also addresses the limitation of Q&A websites that they do not offer support for exploiting their data in a team-working context [43]. By enabling developers to link imported code snippets to their documents via language-independent annotations, *Seahawk* helps developers share documents with their teammates [321]. The evaluation experiments were performed on 35 exercises from Java training courses [320]. The results were generally promising. Although the tool might not always suggest useful documents, it sometimes aided developers with surprising insights.

*WordMatch* and *SnipMatch* [386] are two search tools for helping developers integrate crowdsourced code snippets. *WordMatch* provides an end-user programming environment that enables users (without programming experience) to generate direct answers to search queries. *SnipMatch* is an Eclipse plugin built on *WordMatch* that retrieves customised, ranked source code snippets, based on current code context and the developer's search query.

Souza et al. [94] also aimed to use crowd knowledge from *StackOverflow*, but focused on proposing a ranking approach for potential solutions. The ranking strategy is based on two factors, including the quality of question-answer pairs and the textual similarity of the pairs regarding the developer's query. Experiments were performed on three programming topics. The results demonstrated that at least one suggested question-answer pair is helpful for 77.14% of the evaluated activities.

Amann et al. [29] investigated on using crowd knowledge for method-call recommendations. Crowd knowledge was collected from multiple developers' implicit feedback on their context-sensitive usage of the APIs. Collaborative filtering techniques were employed for recommending method calls based on such feedback knowledge.

Bruch [67] proposed the idea of *IDE 2.0* (based on the concept of *Web 2.0*). Bruch showed how crowd knowledge can help improve multiple functions such as API documentation, code completion, bug detection and code search. Evaluations were performed on each of the proposed tools, revealing that the concept of *Web 2.0* can be

leveraged to improve the developer's IDE.

Fast et al. [109] conducted a study that echoes the idea of *IDE 2.0*. However, it focused on codifying emergent programming behaviour. By building a knowledge-based named *Codex*, which contained more than three million lines of popular Ruby code, novel data driven interfaces were constructed. For example, *Codex* was used for detecting unusual code that may contain a bug, annotating popular programming idioms identified by the system and generating utility libraries that capture emerging programming practice. According to Fast et al. [109], limitations of the current version of the proposed tool may include the adoption of GitHub, the only source of training data, which may introduce open sourced code with low quality.

Using the crowd knowledge to find common examples from the web, shares similarities with work on automatic harvesting of realistic test cases from the web-based systems [15, 60]. As with the potential for the combination of genetic improvement and social recommenders, this similarity also points to the possibility of hybridise versions that harvest such information from a combination of crowd and web for testing purposes.

#### Crowdsourcing for Software Evolution and Maintenance

Software evolution and maintenance are among the earliest areas that have benefited from the application of crowdsourcing. A series of studies have investigated the potential of crowdsourced software evolution and maintenance [21, 24, 45, 75, 107, 135, 140, 155, 181, 237, 244, 262, 307, 309].

#### Crowdsourced Software Evolution

Formal or automated verification methods may fail to scale to large software systems [45]. To help scalability, a market-based software evolution mechanism was proposed by Bacon et al. [45]. The goal of the mechanism is not to guarantee the absolute 'correctness' of software, but rather to economically fix bugs that users care about most. The proposed mechanism lets users bid for bug fixes (or new features) and rewards the bug reporters, testers and developers who respond.

Software adaptation aims to satisfy users' dynamic requirements. However, context is

difficult to capture during the software design phase, and it is a challenging task to monitor context changes at runtime. Ali et al. [20] proposed *Social Sensing* to leverage the wisdom of the end users and used them as monitors for software runtime adaptation. This technique may help software designers (and their systems) to capture adaptation drivers and define new requirement and contextual attributes through users' feedback. A follow-up work of *Social Sensing* is *Social Adaptation* [21], in which several techniques (such as the goal model) for realising social sensing were further discussed. Also, evaluation of the proposed framework was performed on a socially adaptive messenger system. He et al. [155] proposed a 'suggestion model' to encourage crowd users to become more closely involved in commercial software runtime adaptation. A prototype and several adaptation strategies were introduced in this study. Challiol et al. [73] proposed a crowdsourcing approach for adapting mobile web applications based on client-side adaptation.

Nebeling and Norrie [284, 285] presented an architecture and visual supporting tools for facilitating crowdsourced web interface adaptation. Design and technical challenges when applying the crowdsourcing model, especially for quality control, were discussed. A tool named *CrowdAdapt* [289] was further implemented and evaluated. Experimental results showed the tool's capability in leveraging crowd users for generating flexible web interfaces.

In order to tackle the 'bloat' issue in enterprise applications, Akiki et al. [19] focused on utilising crowdsourcing for UI adaptations. Their proposed approach is based on model-driven UI construction which enables the crowd to adapt the interfaces via an online editing tool. A preliminary online user study pointed to promising results on usability, efficiency and effectiveness of the approach.

Users may become overwhelmed by the number of choices offered by software systems. In order to provide customised configuration dialogs to users, Hamidi et al. [140] proposed to extract configuration preferences from a crowd dataset. The optimised configuration dialogs were formed using a Markov Decision Process. When constructing customised dialogs, configuration decisions can be automatically inferred from knowledge elicited in previous dialogs. The evaluation of the method was performed on a

Table 2.10: Crowdsourced evaluation for software engineering research

| Ref. | SE Task | Size | Crowd | Platform | Effort |
|---|---|---|---|---|---|
| [123] | Fault localisation | 65 | Developers | AMT | 1,830 judgements |
| [354] | Code smell impact evaluation | 50 | End users programmers | AMT | 160 HIT responses |
| [154] | IDE enhancement | 13 | Students | Workshop | 39 person-hours |
| [124] | Patch maintainability | 157 | Developers | Campus, AMT | 2,100 judgements |
| [15] | Readability evaluation on test input strings | 250 | Developers | CrowdFlower | 8 questions per task, 250 responses |
| [356] | Code smell impact evaluation | 61 | End user programmers | AMT | 366 task responses |
| [355] | Survey on code search habits | 99 | Developers | Campus, AMT | 10 questions per survey |
| [109] | Code annotation | - | Developers | oDesk | 500 code snippets' evaluation |

Facebook dataset collected from 45 student users. Experimental results indicated that the proposed method could help users to reduce configuration steps by 27.7%, with a configuration prediction precision of 75%.

**Crowdsourcing for Software Documentation** Software documentation plays a crucial role in program understanding. Previous studies have pointed out that inaccurate or insufficient documentation is a major cause of defects in software development and maintenance [90, 186, 374]. Several researchers have investigated crowdsourcing models to enhance software documentation [50, 75, 181, 307, 309].

Jiau and Yang [181] conducted an empirical study based on StackOverflow to reveal the severe uneven distribution of crowdsourced API documentation. To deal with the inequality, a reuse method based on object inheritance was proposed. An empirical evaluation was performed on three Java APIs: GWT, SWT and Swing. The results confirmed the feasibility of the documentation reuse methods with improved documentation quality and coverage.

Parnin et al. [307] conducted a similar empirical study, but with a focus on investigating the coverage and dynamics of API documentation supported by StackOverflow. Three APIs including the Java programming language, GWT and Android, were studied. The results showed that the crowd was able to generate rich content with API usage examples and suggestions. For example, for Android, 87% of its classes were covered

by 35,000 developer contributed questions and answers. However, since the study is based on a single Q&A platform, there may exist issues in generalising the findings.

Chen and Zhang [75] also studied crowd knowledge for API documentation. Documentation reading and searching behaviours were recorded for extracting question and answer pairs. Frequently asked questions were maintained for generating expanded API documentation automatically.

Pawlik et al. [309] conducted a case study on crowdsourced software documentation for *NumPy* (a Python library for scientific computing). The case study highlighted aspects that need to be considered when applying crowdsourcing for software documentation, e.g., technical infrastructure, stylistic instruction and incentive mechanism.

**Crowdsourcing for Software Localisation** Software localisation is also relevant to 'software internationalisation' or 'globalisation' [244], such as tailoring the natural language output from systems for each country in which they are deployed. Localisation may be an important factor for the adoption and success of international products [105]. Research on utilising crowdsourcing for software localisation [107, 135, 244, 262] aim to reduce the cost and time-to-market periods of the traditional developer-based localisation process.

Exton et al. [107] first proposed the idea to use crowdsourcing for software localisation. Manzoor [244] developed a prototype for crowdsourced software localisation. An Action-Verification Unit method, together with a quality-oriented rewarding system, was proposed for quality control. The preliminary evaluation results showed that outcomes with acceptable quality can be delivered by the crowd. Gritti [135] also worked on a similar project and established a prototype system for crowdsourced translation and software localisation.

#### Crowdsourcing for Other Software Engineering Activities

Crowdsourcing has also been applied to support other software engineering activities, such as software security and privacy analysis [17, 36, 69, 169, 230, 231, 306, 330, 339], software end user support [80, 81, 82] and software ideation [102, 175, 176, 200].

Many previous studies have demonstrated that crowdsourcing is an effective way to augment software security [17, 36, 69, 306, 330, 339]: Arellano [36] proposed crowdsourced web augmentation, based on the idea that end users are not only beneficiaries of web augmentation scripts, but can also contribute to them. Sharifi et al. [339] implemented a system called SmartNotes for detecting security threats underlying web browsing.

The increasing number of malicious mobile apps makes malware analysis an urgent problem. Burguera et al. [69] presented a novel crowdsourced framework named *Crowdroid* for detecting Android malware. App behaviour traces were collected from real users (in the crowd), and were subsequently used for differentiating malicious or benign apps. The experimental results showed a 100% detection rate in 3 self-written apps. In another real-world app experiment, the detection accuracies were 85% and 100% for two real malware specimens.

Users frequently struggle with reviewing permissions requested by mobile apps. Inappropriately granted permission may cause privacy leaks. Lin [230] collected the permissions granted to mobile apps from a crowd consisting of over 700 mobile phone users. The collected privacy preferences were analysed using clustering algorithms, and the privacy profiles identified to be important were used to provide default permission settings for mitigating user burden. An evaluation, based on three fake apps and the crowd recruited from Amazon Mechanical Turk, indicated the resulting preference models were able to relieve users' burden in choosing privacy settings. Agarwal and Hall [17] introduced a crowdsourced recommendation engine called ProtectMyPrivacy, which detects and deals with privacy leaks for iOS devices. Papamartzivanos et al. [306] introduced a cloud-based architecture which is driven by the crowd for privacy analysis of mobile apps. Ismail et al. [169] proposed a crowd manage strategy for security configuration exploration, aiming to find minimal permission sets that preserve app usability. The experiment conducted via a small crowd of 26 participants demonstrated the efficiency of the proposed strategy and the usefulness of the recommended configurations.

Regarding crowdsourced end user support, Chilana et al. [80, 81, 82] proposed *Lemon-*

*Aid*, a tool for providing contextual help for web applications, enhanced by crowd knowledge. The tool retrieves users' previously asked questions and answers in response to their UI selections on the screen. The evaluation performed on Amazon Mechanical Turk showed that *LemonAid* was able to retrieve at least one user support answer for 90% of the selection behaviours studied, and a relevant answer was likely to be in the top two results. Results from a field study (by deploying *LemonAid* to multiple sites) suggested that over 70% of the end users were likely to find a helpful answer from *LemonAid* and might reuse the support system.

Software engineering research can also benefit from crowdsourcing. It can be used to conduct human studies [15, 123, 124, 154, 354]. We summarised a few studies on using crowdsourced evaluation for software engineering research in Table 2.10. Note that we do not claim to have surveyed such crowdsourced human studies in software engineering research comprehensively, as this is not the focus of this study but it can be a direction for future work. The model can also be employed in organising broadly accessible software engineering contests [86] such as Predictive Models in Software Engineering (PROMISE), Mining of Software Repositories (MSR) and Search Based Software Engineering [145] (SBSE) challenges.

Several authors have anticipated that crowdsourcing will be applied to address more challenges in software engineering research [78, 157, 367].

### 2.2.6 Issues and Open Problems

Despite the extensive applications of crowdsourcing in software engineering, the emerging model itself faces a series of issues that raise open problems for future work. These issues and open problems have been identified by previous studies. However, few research studies have focused on solutions to address these issues.

According to an in-depth industrial case study on TopCoder [351], key concerns including task decomposition, planning and scheduling, coordination and communication, intellectual property, motivation and quality challenges were highlighted as interesting and important challenges.

Several studies are concerned with suggesting potential research topics. Stol and Fitzgerald [353] presented a research framework inspired by the issues identified in the TopCoder case study [351]. It took the perspective of three key stakeholders, i.e., the requester, the platform and the worker. Research questions were proposed for issues identified from the view of each of the three stakeholders. LaToza et al. [208] briefly outlined a series of research questions concerning the division of crowd labour, task assignment, quality assurance and the motivation of the crowd's participation. A follow-up research agenda can be found in the recent paper [207].

In the remainders of this section, we discuss Crowdsourced Software Engineering issues together with relevant work in more detail:

#### Theory and Model Foundations

The use of undefined external workforce differentiates Crowdsourced Software Engineering from conventional software engineering. Existing software development theories and models may no longer apply to this emerging model [191, 192, 245].

In order to better facilitate Crowdsourced Software Engineering, a series of theories and models have been proposed. The first published theoretical model for Crowdsourced Software Engineering is the Metropolis Model proposed by Kazman and Chen [191, 192], who argued that classical software development models such as the waterfall model, the spiral model and the more recent agile models are not suitable for Crowdsourced Software Engineering.

The Metropolis Model distinguishes three types of roles, i.e., the platform (referred to as *kernel*), applications built on the *kernel* (referred to as *periphery* ), and the end users (referred to as *masses*). Seven principles of the model were introduced for managing crowdsourced development.

Saxton et al. [331] subsequently analysed 103 crowdsourcing websites and provided a taxonomy of nine crowdsourcing models. Among them, the Intermediary Model and the Collaborative Software Development Model support Crowdsourced Software Engineering.

Tsai et al. [366] summarised the commonalities in different Crowdsourced Software Engineering processes and proposed an architecture for cloud-based software crowdsourcing. The architecture specifies a management web interface for the requesters, a series of development tools for online workers, worker ranking and recommendation tools provided by the platform, collaboration tools for multiple stakeholders, a repository for software assets and a cloud-based payment system.

A few studies have also considered game theoretic crowd formulations to understand competition among crowd developers [163, 391, 395]. Wu et al. identified the 'min-max' (defence-offence) nature of crowdsourced software development competitions and argued that the nature contributes to the quality and creativity of the produced software [391]. Hu and Wu [163] proposed a game theoretic model for analysing the competition behaviours among TopCoder developers. The conclusions of this paper were drawn based on theoretical analysis, e.g., Nash equilibria computation, without empirical evaluation, so the applicability of the model remains to be analysed in future work.

#### Task Decomposition

Crowdsourced complex tasks lead to heavy workloads and require dedicated resources. With inherently high skill barriers, the number of potential workers will inevitably become limited. In order to increase parallelism and to expand the qualified labour pool, it is essential to decompose software engineering tasks into self-contained, smaller or even micro pieces. However, software engineering tasks are often concerned with specific contexts, for which decomposition may be non-trivial. Several studies focused on this decomposition problem.

LaToza et al. [210] developed an approach for decomposing programming work into micro-tasks. The method breaks down a single higher level task into multiple lower level tasks iteratively, and coordinates work by tracking changes linked to artefacts. A platform called *CrowdCode* [209] was implemented to support their proposed method. The evaluation was performed on a crowd of 12 developers and the results indicated that the approach had an 'overhead issue' which led to a potentially lower productivity compared to the traditional development methods. LaToza et al. [211] also proposed

to decontextualise software development work as part of decomposition. Three types of development work including programming, debugging and design were discussed regarding their decontextualisation.

As discussed in the crowdsourcing applications for software testing and verification (Section 2.2.5), two previous studies also offered decomposition approaches: Chen and Kim [78] decomposed the test generators' complex constraint solving and object mutation problems into small puzzles, which can be solved by crowd labours. Schiller and Ernst [332] proposed an online IDE for verification named *VeriWeb*, which can decompose the verifiable specifications task into manageable sub-problems.

#### Planning and Scheduling

The highly heterogeneous nature of crowd labour necessitates careful planning and scheduling.

Tran-Thanh et al. [365] proposed a bounded multi-armed bandit model for expert crowdsourcing. Specifically, the proposed $\varepsilon$-first algorithm works in two stages: First, it explores the estimation of workers' quality by using part of the total budget; Second, it exploits the estimates of workers' quality to maximise the overall utility with the remaining budget. The evaluation of the proposed algorithm was based on empirical data collected from oDesk. The results indicated that the algorithm was able to outperform related state-of-the-art crowdsourcing algorithms by up to 300%.

Tung and Tseng [367] focused on using crowd resources effectively to support collaborative testing and treated the problem as an (NP-Complete) job assignment problem. They proposed a greedy approach with four heuristic strategies. To evaluate the proposed model, a Collaborative Testing System (COTS) was implemented. Experimental results showed the system was able to generate the average objective solution within approximately 90% of the optimal solutions. When applied to a real-time crowd testing environment, the system was able to save 53% of the test effort.

In some open call formats such as online competition, the tasks are given to unknown developers rather than assigned to specific crowd participants. In such cases, the de-

velopers cannot be directly scheduled but may be optimised using recommendation techniques to guide them to work on their most suitable tasks. Mao et al. [246] employed a content-based technique to recommend developers for crowdsourced software development tasks. The approach learns from historical task registration and winner records to automatically match tasks and developers. Experimental results on TopCoder datasets indicated the recommendation performance was promising in both accuracy (50%-71%) and diversity (40%-52%).

Estimating the appropriate number of crowd developers and delivery time for Crowdsourced Software Engineering tasks is an important yet challenging problem. To date, very limited work has been done in this research area. Mäntylä and Itkonen [243] studied how the crowd size and allocated time can affect the performance of software testing. Their results, conduced on 130 students, indicated that multiple crowd workers under time pressure had 71% higher effectiveness (measured by the number of detected bugs) than the single workers without time pressure. The authors suggested that the number of crowd workers for manual testing tasks should be adjusted according to the effectiveness of the mechanisms and tools for detecting invalid and duplicate bug reports.

To guarantee sufficient high participation levels in Crowdsourced Software Engineering tasks, Wang et al. [377] proposed a framework to support crowdsourcing systems in their recruitment of participants with domain knowledge for requirements acquisition. The framework was established based on the observation that crowd workers with similar domain knowledge tend to cluster in particular spatio-temporal regions. The feasibility of this framework was demonstrated by a theoretical study and a simulation experiment.

#### Motivation and Remuneration

Motivation is viewed as a critical factor for the success of a software project [51, 58, 340]. For crowdsourced software projects, developers without proper motivation may not be able to make consistent contributions, while inappropriate remuneration may lead to low capital efficiency or task starvation. Varshney [370] demonstrated that player mo-

tivation is essential for driving participation and ensuring a reliable delivery platform. Based on a study from IBM's internal crowdsourced software development system — *Liquid*, several intrinsic, extrinsic, and social motivation factors were identified. Developer participation was found to follow a power-law distribution. A momentum-based generative model and a thermodynamic interpretation were used to describe the observed participation phenomena.

Mao et al. [245] proposed 16 cost drivers for training empirical pricing models to meet crowd developers' monetary remuneration. Specifically, the development type (upgrade or new development) of the task, the number of component specifications, the number of sequence diagrams of the design and the estimated size of the task were considered as significant factors that impact the remuneration. Based on the identified cost drivers, nine predictive pricing models were trained using popular machine learning algorithms. Evaluation on 490 TopCoder projects indicated that high prediction quality was achievable.

Leimeister et al. [200] investigated the motivation of participants for IT-based idea competitions. Incentives such as organiser's appreciation, prizes and expert knowledge were highlighted in this study. Olson and Rosacker [299] discussed the motivation for participating in crowdsourcing and open source software (OSS) development. The element of altruism was considered to be important in motivating participation in both OSS and crowdsourced software development. Ramakrishnan and Srinivasaraghavan [325] presented intrinsic motivational factors (e.g., skill variety and peer pressure) and extrinsic motivational factors (e.g., monetary reward and recognition) among students in a crowdsourced programming task context. A controlled experiment was performed to show the viability of employing a captive university crowd for software development.

#### Quality Assurance

Crowd labour is transient and workers vary in expertise and background. The use of such an undefined workforce inherently raises quality questions for crowdsourcing in general [23, 168, 402] as well as Crowdsourced Software Engineering [208, 220, 328, 351].

Li et al. [220] identified 23 quality factors for crowdsourced software development from the perspective of platform and project, based on an empirical study of TopCoder. Four important aspects were identified in order to improve crowdsourced software quality, including the prosperity level of the platform, the scale of the task, the participants' skill levels and the design quality of the task.

Saengkhattiya et al. [328] investigated how crowdsourcing companies deal with the quality assurance challenge by conducting interviews with four companies: Microworkers, Clickchores, Microtask and TopCoder. Ten diverse methods for managing quality were identified, such as ranking/rating, reporting spam, reporting unfair treatment, task pre-approval, and skill filtering.

Tajedin and Nevo [358] built a 'success model' of crowdsourced software development, which contains three high-level determinants, namely the project characteristics, the crowd composition and the stakeholder relationship. The model was proposed based on the analysis of related studies on the success of information systems, OSS development and general software development.

Much of the work on quality assurance remains to be fully evaluated, leaving rigorous evaluations of Crowdsourced Software Engineering quality assurance as a pressing topic for future work.

### 2.2.7 Opportunities on Hybrid Crowdsourced Software Engineering

The Crowdsourced Software Engineering solutions surveyed in this chapter typically concern the substitution of a crowdsourced activity for an existing (non-crowdsourced) activity. In this regard, the solution is either crowdsourced or not crowdsourced, with a sharp 'binary divide' between the two kinds of activity. We envisage this binary divide becoming blurred as Crowdsourced Software Engineering achieves greater penetration into the research and practitioner communities.

This blurring of the distinction between traditional and crowdsourced activities will lead to a further development of Hybrid Crowdsourced Software Engineering. Tools such as *CrowdBlaze* (Section 2.2.4) already offer a form of hybridisation between crowdsourc-

ing and automated software testing, where the crowd is introduced when automated testing is not sufficient in covering functionalities of the subjects. While bug fix recommendation tools such as *HelpMeOut* could be augmented with genetic improvement (as mentioned in Section 2.2.5) .

Hybrid Crowdsourced Software Engineering will require new processes and methodologies that feedback crowdsourced knowledge into software development process (as it proceeds) and that feed software development information back to the crowd. The growth in the use of app stores as a platform for software deployment and review [79, 138, 147, 304], is already providing a kind of Hybrid Crowdsourced Software Engineering. The review mechanisms implemented by app stores already resemble a channel of communication between the users (a crowd) and an app's developers. We envisage greater deployment, extension and development of such crowdsourced software deployment, review and feedback infrastructures.

Existing work on crowdsourced software testing usually directly outsources testing tasks to crowd individuals, where the collective crowd intelligence is not utilised. In Chapter 5 of this thesis, we further investigate Hybrid Crowdsourced Software Engineering in mobile testing, where the mobile testing is partially crowdsourced and partially automated. By collecting and learning from the partially crowdsourced mobile manual testing traces, useful test input patterns are extracted to assist automated mobile test generation.

### 2.2.8 Summary

In this chapter, we have briefly summarised existing bodies of literature in automated mobile testing for Android and JavaScript applications. We have also comprehensively analysed related work on the use of crowdsourcing in software testing and other software engineering activities. We have the following conclusions covering the existing literature:

- Previous work on automated mobile testing covers several important testing objectives, such as coverage, test sequence length, execution time, readability and

replicability, yet none optimises these competing objectives simultaneously or provides a set of Pareto-optimal tradeoff solutions.

- Current mobile testing techniques still fail to outperform random testing in terms of coverage and fault detection capability.
- The state-of-the-art automated Android testing techniques only achieves relatively poor statement coverage (less than 50%) [84].
- An increasing number of papers have been published on using crowdsourcing to support software engineering activities, yet little work has been done on using crowdsourcing to support test automation.

# Chapter 3

# Sapienz: Multi-objective Automated Android Testing

This chapter introduces SAPIENZ, an Android testing approach that uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising test sequence length, while simultaneously maximising coverage and fault revelation. SAPIENZ combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation. SAPIENZ significantly outperforms (with large effect size) both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length. When applied to the top 1,000 Google Play apps, SAPIENZ found 558 unique, previously unknown crashes. So far we have managed to make contact with the developers of 27 crashing apps. Of these, 14 have confirmed that the crashes are caused by real faults. Of those 14, six already have developer-confirmed fixes.

## 3.1 Introduction

There are over 2.6 million apps available from the Google Play marketplace, as of January 2017 [8]. For developed internet markets such as the US, UK and Canada,

mobile app usage now dominates traditional desktop software usage [89]. Unfortunately, testing technology has yet to catch up, and software testers are faced with additional problems due to device fragmentation [4], which increases test effort due to the number of devices that must be considered. According to a study on mobile app development [183], mobile app testing still relies heavily on manual testing, while the use of automated techniques remains rare [198].

Where test automation does occur, it typically uses Google's Android Monkey tool [133], which is currently integrated with the Android system. Since this tool is so widely available and distributed, it is regarded as the current state of practice for automated software testing [240]. Although Monkey automates testing, it does so in a relatively unintelligent manner: generating sequences of events at random in the hope of exploring the app under test and revealing failures. It uses a standard, simple-but-effective, default test oracle [49] that regards any input that reveals a crash to be a fault-revealing test sequence.

Automated testing clearly needs to find such faults, but it is no good if it does so with exceptionally long test sequences. Developers may reject longer sequences as being impractical for debugging and also unlikely to occur in practice; the longer the generated test sequence, the less likely it is to occur in practice. Therefore, a critical goal for automated testing is to find faults with the *shortest possible* test sequences, thereby making fault revelation more actionable to developers.

Exploratory testing is "simultaneous learning, test design, and test execution" [12], that can be cost-effective and is widely used by industrial practitioners [44, 170, 188] for testing in general. However, it is particularly underdeveloped for mobile app testing [171, 172]. Although there exist several test automation frameworks such as Robotium [11] and Appium [5], they require human-implemented scripts, thereby inhibiting full automation.

We introduce Sapienz, the first approach offering multi-objective automated Android app exploratory testing that seeks to maximise code coverage and fault revelation, while minimising the length of fault-revealing test sequences. Our goal is to produce an entirely automated approach that maximises fault revelation with short test sequences.

The key insight in our approach is that minimising test sequence length and maximising other objectives can be combined in a Pareto-optimal multi-objective search-based approach to Android testing. By using Pareto optimality, we do not sacrifice longer test sequences, when they are the only ones that find faults, nor where they are necessary to achieve higher code coverage. Nevertheless, through its use of Pareto optimality, SAPIENZ progressively replaces such longer sequences with shorter test sequences when equally good. The paper makes the following primary contributions:

**1) The Sapienz approach:** the paper introduces the first Pareto multi-objective approach to Android testing, combining techniques used for traditional automated testing, adapting and extending them for Android testing. The approach combines random fuzzing, systematic and search-based exploration, string seeding and multi-level instrumentation, all of which have been extended to cater for, not only traditional white box coverage (which we term 'skeletal coverage'), but also Android UI coverage (which we term 'skin coverage').

**2) Experimental results:** we present the results of two systematic experimental studies on open-source real-world Android apps. The first uses the 68 apps from an Android benchmark suite [84], while the second uses a controlled random sample of 10 apps from the entire F-Droid suite, for which SAPIENZ always outperforms both Dynodroid and Monkey, statistically significantly and with large effect size in 24 out of 30 cases.

**3) The tool, Sapienz:** a practical Android testing tool SAPIENZ, which we make publicly available[1].

**4) Demonstration of usefulness:** an empirical study of the practical usefulness of the technique on the top 1,000 Google play apps. SAPIENZ found 558 unique crashes. The crashing behaviour has been verified on real Android devices (as well as Android emulators). At the time of writing, we have started reporting these to the developers, and 14 have been confirmed to be genuine, previously undetected, faults, 6 of which have already been confirmed as fixed by their developers. Since these are the most popular apps in current use, they will likely have been thoroughly tested, not merely

[1] http://github.com/Rhapsod/sapienz

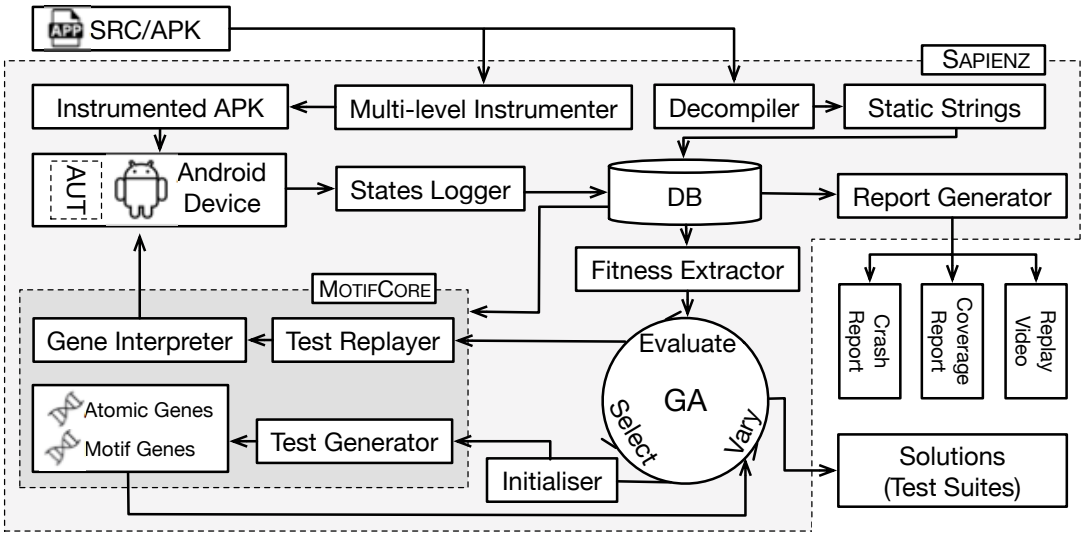



Figure 3.1: Sapienz workflow

by their developers, but also by their many (hundreds of thousands of) users. These results demonstrate that SAPIENZ is a practical tool for Android developers as well as for researchers. This is the first Android app testing work to report a large-scale evaluation on popular Google Play apps with developer-confirmed real-world faults.

## 3.2 The Sapienz Approach

We first outline the workflow used by our approach. Then we provide component summaries of our evolutionary algorithm. The exploration strategy and app analysers of SAPIENZ are described in Sections 3.2.2 and 3.2.3 respectively.

SAPIENZ' overall workflow is depicted in Figure 6.1. SAPIENZ takes the app under test as the input, which can be either the app's source code or its binary APK file. SAPIENZ produces a set of Pareto-optimal solutions (test suites) and detailed test reports regarding the whole evolutionary testing process, as the output.

SAPIENZ starts by instrumenting the app under test, which can be achieved in a white box, grey box or black box manner as follows: When the app's source code is available, SAPIENZ uses fine-grained instrumentation at the statement-level (white box). By contrast, should it turn out that only the binary APK file is available (as is often the case in real-world, industrial-strength Android testing scenarios), SAPIENZ uses undexing

and repacking to instrument the app at method-level (grey box). However, where the developers disallow repackaging (as is common for commercial apps), SAPIENZ uses a non-invasive activity-level 'skin' coverage, which can always be measured (black box).

SAPIENZ extracts statically-defined string constants by reverse/engineering the APK. These strings are used as inputs for seeding realistic strings into the app, which has been found to improve the performance of search-based software testing techniques for web based testing [26], and traditional application testing [118], and also to improve realism [60], but has not previously been used in Android testing. Test sequences are generated and executed by the MOTIFCORE component, which combines random fuzzing and systematic exploration, which corresponds to two types of genes: the low-level *atomic genes* and the high-level *motif genes*.

SAPIENZ' multi-objective search algorithm generates the initial population via MOTIFCORE's *Test Generator*. During the genetic evolution process, genetic individuals are assigned to the *Test Replayer* when evaluating individual fitnesses. The individual test scripts are further decoded into executable Android events by the *Gene Interpreter*, which communicates with the the Android device via the Android Debugging Bridge (ADB). The *States Logger* monitors the execution states (e.g., covered activities, crashes) of the App Under Test (AUT) and produces measurement data for the *Fitness Extractor* to calculate the fitnesses. A set of Pareto-optimal solutions and test reports are generated at the end of the search.

### 3.2.1 Multi-objective Search Based Testing

Algorithm 3.1 presents SAPIENZ' top-level algorithm. SAPIENZ optimises for three objectives: code coverage, sequence length and the number of crashes found, using a Pareto-optimal Search Based Software Engineering (SBSE) approach [143, 148].

Each executable test suite $\vec{x}$ for the AUT is termed as a *solution* and a *solution* $\vec{x_a}$ is

**Algorithm 3.1:** Overall algorithm of SAPIENZ

```
Input: AUT A, crossover probability p, mutation probability q, max generation g_max, execution time t
Output: UI model M, Pareto front PF, test reports C
M ← K_0; PF ← ∅; C ← ∅;                                   ▷ initialisation
generation g ← 0;
boot up devices D;                                          ▷ prepare app exerciser
inject MotifCore into D;                                    ▷ for hybrid exploration (see §3.2.2)
static analysis on A;                                       ▷ for string seeding (see §3.2.3)
instrument and install A;
initialise population P;                                    ▷ hybrid of random and motif genes
evaluate P with MotifCore and update (M,PF,C);
while g < g_max and ¬ timeout(t) do
    g ← g+1;
    Q ← wholeTestSuiteVariation(P, p, q);                   ▷ see Algorithm 3.2
    evaluate Q with MotifCore and update (M,PF,C);
    F ← ∅;                                                  ▷ non-dominated fronts
    F ← sortNonDominated(P ∪ Q, |P|);
    P' ← ∅;                                                 ▷ non-dominated individuals
    for each front F in F do
        if |P'| ≥ |P| then break;
        calculate crowding distance for F;
        for each individual f in F do
            P' ← P' ∪ f;
    P' ← sorted(P', ≺_c);                                   ▷ see equation 3.3 for operator ≺_c
    P ← P'[0 : |P|];                                        ▷ new population
return (M,PF,C);
```

*dominated* by solution $\vec{x_b}$ ($\vec{x_a} \prec \vec{x_b}$) according to a fitness function if and only if:

$$\begin{aligned} &\forall i = 1, 2, ..., n,\ f_i(\vec{x_a}) \leq f_i(\vec{x_b}) \wedge \\ &\exists j = 1, 2, ..., n,\ f_j(\vec{x_a}) < f_j(\vec{x_b}) \end{aligned} \tag{3.1}$$

A *Pareto-optimal set* consists of all *Pareto-optimal* solutions (belonging to all solutions $X_t$), which is defined as:

$$P^* \triangleq \{\vec{x^*} \mid \nexists \vec{x} \in X_t, \vec{x} \prec \vec{x^*}\} \tag{3.2}$$

SAPIENZ' search-based approach uses NSGA-II to build successively-improved Pareto-optimal sets, seeking new dominating test vectors. NSGA is a series of widely-used multiobjective evolutionary search algorithms, popular in SBSE research [148, 271, 301, 302, 400]; the details of NSGA-II can be found elsewhere [95].

At the end of search, testers can choose any test suites of interest from the Pareto-optimal set generated by SAPIENZ. In addition to the Pareto-optimal solution, SAPIENZ also produces an all-crash-test-suite with a set of videos for each crashing scenario. This crashing test suite is generated by an archive operator which stores any crash found

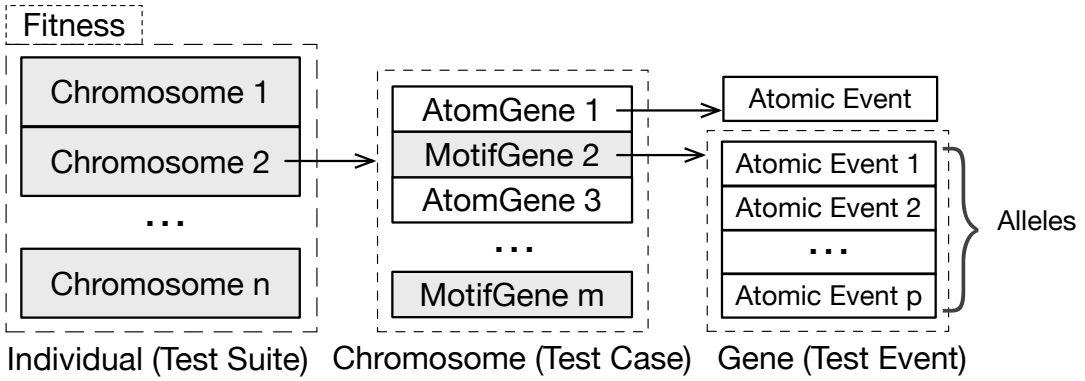



Figure 3.2: Genetic individual representation

during the search process.

**SBSE representation:** SAPIENZ performs the whole test suite evolution [26, 115] thus each individual corresponds to a test suite. The representation of an individual test suite generated by SAPIENZ is illustrated in Figure 3.2. SAPIENZ generates a set of these individual test suites, which corresponds to a population of individuals in the evolutionary algorithm. Each individual consists of several chromosomes (test sequences $\langle T_1, T_2, ..., T_m \rangle$) and each chromosome contains multiple genes (test events $\langle E_1, E_2, ..., E_n \rangle$), which consist of a random combination of *atomic* and *motif genes*. An *atomic gene* triggers an atomic event $e$ that cannot be further decomposed, e.g., press down a key, while a *motif gene* is interpreted as a series of *alleles* (atomic events $\langle e_1, e_2, ..., e_p \rangle$).

**SBSE variation operator:** We define a *whole test suite variation operator* to manipulate individuals. The operator is depicted in Algorithm 3.2: It applies one of the finer-grained *crossover*, *mutation* and *reproduction* operators on each individual (at test suite level). SAPIENZ' inter-individual variation is achieved by using a uniform set element *crossover* among individuals (test suites). The inner-individual variation is manipulated by a more complex *mutation* operator. Since each individual is a test suite containing several test cases, the operator first randomly shuffles test case orders and then performs a single-point crossover on two neighbouring test cases with probability $q$, where the prior shuffle operation aims to improve crossover diversity. Subsequently, the more fine-grained test case mutation operator shuffles the test events within each test case with probability $q$, by randomly swapping event positions. Although atomic

events include (mutable) parameters, we choose instead to mutate the execution order of the events, thereby reducing the complexity of the variation operator. Mutants are possible to operate on new GUI widgets not exercised by *any* initial test case, because the timing and order of the operations are mutated. The *reproduction* operator simply leaves a randomly chosen individual unchanged.

**SBSE selection:** We use the *select* operator from NSGA-II [95], which defines a crowding-distance-based comparison operator $\prec_c$. For two test sequences $\vec{a}$, and $\vec{b}$. We say $\vec{a} \prec_c \vec{b}$ if and only if:

$$\vec{a}_{rank} < \vec{b}_{rank} \vee (\vec{a}_{rank} = \vec{b}_{rank} \wedge \vec{a}_{dist} > \vec{b}_{dist}) \tag{3.3}$$

This selection favours test sequences with smaller non-domination rank and, when the rank is equal, it favours the one with greater crowding distance (less dense region).

**SBSE fitness evaluation:** SAPIENZ evaluates the fitness value of each individual by applying the test suite represented by the individual to the app under test, on an Android emulator or a real device. The fitness value of an individual is recorded as a triple for each of the objectives: coverage, length of the test and number of revealed crashes. Specifically, since each individual denotes a test suite which contains multiple test cases, the coverage (and also for length and the number of revealed crashes) is based on the accumulated value for all test cases in the individual.

SBSE Fitness evaluation can be time-consuming, but it is fortunately also embarrassingly parallel [39, 71, 270, 401]. Therefore, in order to achieve time-efficient search, SAPIENZ supports parallel fitness evaluation, assigning individuals to multiple fitness evaluators, which may run on distributed devices (a single multicore machine was used in our evaluation, when comparing SAPIENZ with other techniques).

### 3.2.2 Exploration Strategy

Android apps can have complex interactions between the events triggerable from the UI, and the states reachable and consequent coverage achieved. In manual testing, the

**Algorithm 3.2:** The whole test suite variation operator

```
Input: Population P, crossover probability p, mutation probability q
Output: Offspring Q
Q ← ∅;
for i in range(0, |P|) do
    generate r ∼ U(0, 1);
    if r < p then                                   ▷ apply crossover
        randomly select parent individuals x1, x2;
        x'1, x'2 ← uniformCrossover(x1, x2);
        Q ← Q ∪ x'1
    else if r < p + q then                          ▷ apply mutation
        randomly select individual x1;
        ▷ vary test cases within the test suite x1
        x ← shuffleIndexes(x1);
        for i in range(1, |x|, step 2) do
            generate r ∼ U(0, 1);
            if r < q then
                x[i − 1], x[i] ← onePointCrossover(x[i − 1], x[i]);
        ▷ vary test events within the test case x[i]
        for i in range(0, |x|) do
            generate r ∼ U(0, 1) ;
            if r < q then
                x[i] ← shuffleIndexes(x[i]);
        Q ← Q ∪ x
    else Q ← Q ∪ (randomly selected x1); ▷ apply reproduction
return Q;
```

testers' knowledge can be deployed to explore such complex interactions [172]. However, for automated testing, some other way to handle complex interactions has to be found. Simple approaches to automated Android testing use only atomic events. Even with combinations of such events, the lack of state and context awareness, makes it difficult to discover complex interactions. This may be one reason why many research tools were found to under-perform by comparison with Monkey in the benchmark study conducted by Choudhary et al. [84].

To address this issue, SAPIENZ uses *motif patterns*, which collect together patterns of lower level events, found to be good at achieving higher coverage. *Motif genes* are based on the UI information available in the current view, which is widget-based for Android apps. *Motif genes* work together to perform behavioural usage patterns on the app, e.g, fill all input fields in the current view and submit.

This is achieved by pre-defining patterns to capture testers' experience regarding complex interactions with the app. The *motif gene* is inspired by how a DNA motif works: A DNA motif is a short sequence pattern that has a biological function. Motifs are combined with atomic sequences so that, together, they can express the overall DNA

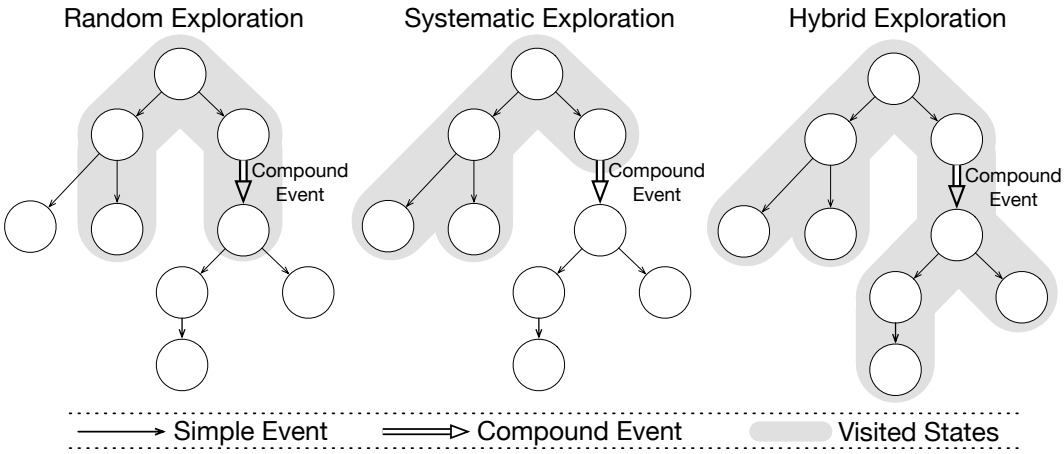



Figure 3.3: Hybrid exploration strategy

function. In our case, our *motif genes* seek to achieve high-level functions (by defining patterns) and to work together with *atomic genes* to achieve higher test coverage. As we explain below, in Section 3.2.4, our evaluation of SAPIENZ relies solely upon a single obvious, default, generic *motif gene*, to avoid any risk of experimenter bias. However, in future work, we may learn motifs from captured human-led test activities.

**Hybrid exploration:** *Atomic genes* and *motif genes* are complementary (see Figure 3.3), so SAPIENZ combines them to form hybrid sequences of test events. Random exploration may (randomly) manage to cover unplanned UI states for compound events (of which consists of a random combination of atomic events), but may generally achieve low overall coverage. Systematic exploration may achieve good coverage within planned UI state regions, but can be blocked by unplanned compounds. The hybrid strategy used by SAPIENZ is shown in Algorithm 3.3.

### 3.2.3 Static and Dynamic Analysis

SAPIENZ performs two types of analysis: static analysis for string seeding and dynamic analysis for multi-level instrumentation. These two features provide necessary information for SAPIENZ to generate realistic test inputs and to guide the search toward optimal test suites with high test coverages.

**String seeding:** In order to extract statically defined strings, SAPIENZ first reverse-engineers the APK file. SAPIENZ obtains a list of globally applicable strings from the

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">BookWorm</string>
...
<string name="btnSelectImage">Select from gallery</string>
<string name="btnRetrieveImage">Retrieve from web</string>
<string name="btnGenerateImage">Generate from title</string>
<string name="labelTitle">Title</string>
<string name="labelSubtitle">Subtitle</string>
<string name="labelSubject">Subject</string>
<string name="labelIsbn10">ISBN 10</string>
<string name="labelIsbn13">ISBN 13</string>
<string name="labelAuthors">Author(s) (comma separated)</string>
<string name="labelAuthorsShort">Author(s)</string>
<string name="labelRating">Rating</string>
...
</resources>
```

Figure 3.4: An example Android static string resource file

decompiled XML resource files. One example Android static string file is shown in Figure 3.4. These natural language strings are randomly seeded into the text fields by the MotifCore component, when performing its hybrid exploration. We found this seeding to be particularly useful when testing apps that require a lot of user-generated content, because it enables Sapienz to post and comment in an apparently more human-meaningful way. When the APK file cannot be reverse-engineered successfully, which is a common case for commercial apps, predefined dummy strings[2] will replace the extracted strings from the app.

**Multi-level instrumentation for skeleton and skin coverage:** In order to be practical and useful, an automated Android testing technique needs to be applicable to both open and closed-source apps. To achieve this, Sapienz uses multi-level instrumentation at one or all of the three levels of applicable instrumentation granularity. The coarsest instrumentation granularity is always possible, and is performed through activity/screen interactions to achieve black box testing or 'skin coverage' as we call it, because it only interacts with the 'surface' UI and system actions of the app. Carino and Andrews also use a similar metric based on the change of GUI widgets [72]. We use the term 'skeletal coverage' for the more fine-grained coverages, achieved by grey and white box instrumentation. In some cases, even when source code is unavailable, a finer-grained, grey box coverage is possible at the method level, which we term 'backbone' skeletal coverage. This backbone coverage can be achieved by undexing the APK

[2]In our particular implementation, a single string of '0' is used to ensure that no fields is empty.

**Algorithm 3.3:** The MotifCore exploration strategy

```
Input: AUT A, test sequence T = ⟨E_1, E_2, ..., E_n⟩, random event list R, motif event list O, static
       strings S existing UI Model M and test reports C
Output: Updated (M, C)
1 for each event E in T do
2     if E ∈ R then                                          ▷ handle atomic gene
3         execute atomic event E and update M;
4     if E ∈ O then                                          ▷ handle motif gene
5         currentActivity ← extractCurrentActivity(A)
          uiElementSet ← extractUiElement(currentActivity) for each element w in uiElementSet do
6             if w is EditText widget then
7                 seed string s ∈ S into w;
8             else
9                 exercise w according to motif patterns in E;
10            update M;
11    (a, m, s) ← get covered activities, methods, statements;
12    C ← C ∪ (a, m, s);                                     ▷ update coverage reports
13 if captured crash c then
14    C ← C ∪ c;                                             ▷ update crash reports
15 return (M, C);
```

file, inserting probes and then repackaging the binary file. Of course, where source code is available, we can and do use traditional statement coverage (which we term 'full skeletal coverage'). For such systems we can cover both the 'skeleton and the skin'; white box statement level coverage and black box UI/activity coverage.

### 3.2.4 Implementation

We have implemented the Sapienz tool on top of the Deap framework [112] for multi-objective test suite evolution. Sapienz achieves full skeletal coverage (statement coverage) using EMMA [2] and backbone coverage (method coverage) using ELLA [1]. It calculates skin coverage (activity coverage) by calling Android's own `ActivityManager` for extracting activity/screen information.

For *atomic genes*, the evaluation version of Sapienz supports 10 types of atomic events that originate from Android system source, including `Touch`, `Motion`, `Rotation`, `Trackball`, `PinchZoom`, `Flip`, `Nav` (navigation key), `MajorNav`, `AppSwitch`, `SysOp` (system operations such as 'volume mute' and 'end call'). Regarding *motif genes*, of course, there is a wide range of choices for motif patterns, and we distinguish between those that are generic (applicable to all apps) and those that are bespoke (applicable to only a small homogeneous set of apps). For our evaluation purposes, we resisted the temptation to have any bespoke *motif genes*, since these would require human intuition and

intelligence. Furthermore, we imbued our evaluation version of the SAPIENZ tool with only a single (intuitively obvious) generic *motif gene* that systematically exercises text fields and clickable UI widgets under the corresponding view, which is applicable to all apps. It first seeds strings into all text fields and then attempts to exercise each clickable widget to transfer to the next view. Such a motif pattern might perform appropriate actions in scenarios such as filling in and submitting a form. We used this simple-minded approach for the evaluation version of SAPIENZ, to avoid risking any experimenter bias that might otherwise introduce human ingenuity into the *motif gene* construction process. As a result, the findings reported in the following section can be regarded as lower bounds on the performance of our approach; with a smarter selection of generic motif patterns, results will improve, and would further improve with the construction of bespoke *motif genes* for particular apps.

The SAPIENZ tool generates a set of artefacts for reuse, including reusable *test suites*, detailed *coverage reports* and *crash reports* (with corresponding fault-revealing test cases and automatically captured crash videos as witnesses for the failures induced by test cases).

## 3.3 Evaluation

We evaluate the SAPIENZ approach by conducting three empirical studies on both open-source and popular closed-source Android apps. We investigate whether SAPIENZ can optimise multiple objectives and find previously unknown real faults, within limited (30 minutes per app) execution time on real-world production hardware.

As a sanity check, we first want to establish that we have a reliable experimental infrastructure. This is because there are a number of settings and parameter choices that could affect the results and, as been widely noted in other areas of empirical software engineering [360, 378], the choice of parameter tuning options can have a dramatic effect on results. To ensure reliability, we check that our infrastructure replicates the results previously reported by Choudhary et al. [84].

**RQ0 (Reliable replication):** Does our experimental infrastructure reliably replicate

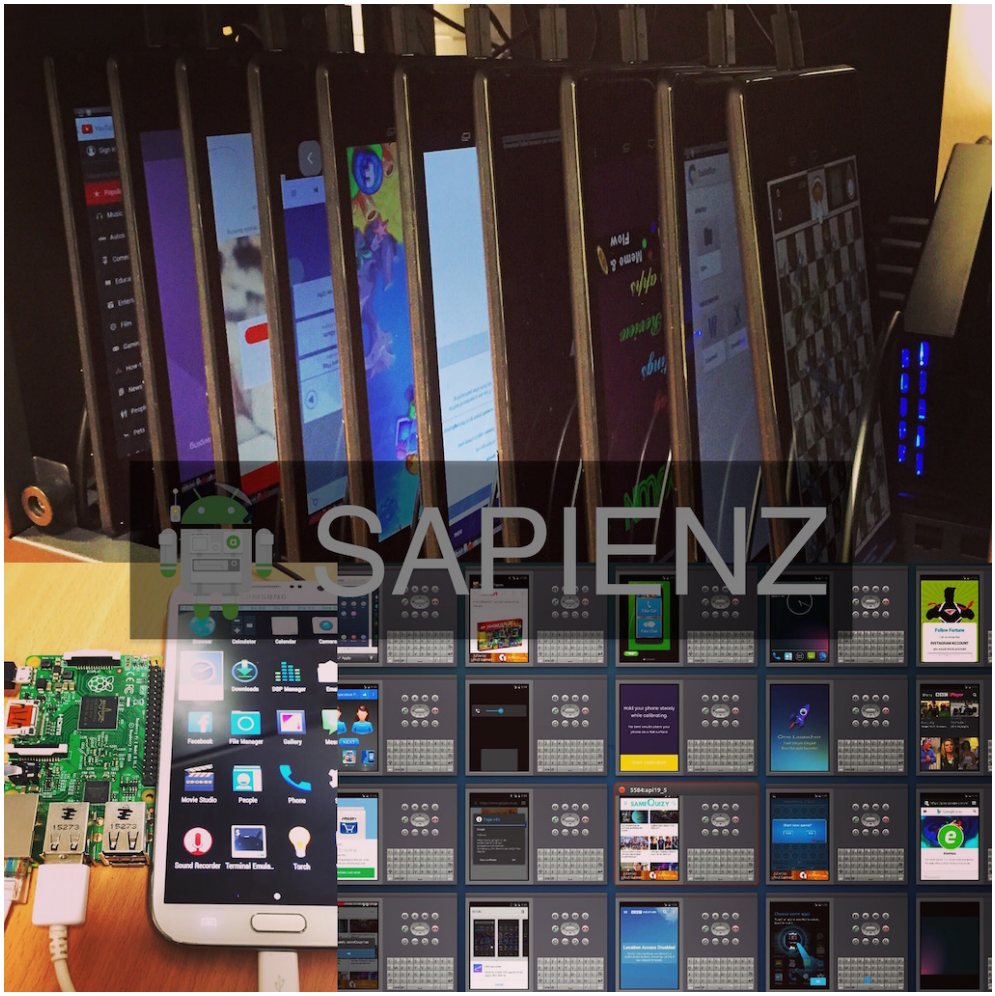



Figure 3.5: Sapienz implementation with a mobile device cluster

the results from the recent thorough study by Choudhary et al. [84]?

We call this RQ0 (rather than RQ1) since it merely establishes that our experimental infrastructure replicates recent results, suggesting that it is reliable for answering the subsequent (novel) questions. A natural question to ask for RQ1, once we have established replication of Choudhary et al. in RQ0, is one that is asked by many other studies [41, 83, 233, 239, 240, 268, 399]: 'what coverage is achieved by the newly proposed technique?'

**RQ1 (Code coverage):** How does the coverage achieved by SAPIENZ compare to the state of the art and the state of practice?

Coverage is one useful indicator, simply because failure to achieve coverage leaves aspects of the app untested. Nevertheless, there is evidence that coverage alone, cannot be

relied upon to indicate test effectiveness [273]. Therefore, our second question focuses on fault detection; regardless of coverage achieved, the effectiveness of any software testing technique should also be assessed by its ability to reveal faults.

**RQ2 (Fault revelation):** How do the faults found by SAPIENZ compare to those found by the state of the art and the state of practice?

SAPIENZ targets coverage, fault revelation and length of fault-revealing test cases. Longer test sequences might achieve higher coverage, but we need to provide short sequences to testers for debugging purposes [34]. Intuitively, shorter sequences are more likely to be attractive and actionable to developers [119, 217]. This motivates RQ3.

**RQ3 (Sequence length):** How does SAPIENZ compare to the state of the art and the state of practice in terms of the length of the fault-revealing test sequences it returns?

We wish to go further in our empirical analysis, because the Choudhary et al. benchmark suite set [84], although an excellent starting point, consists of only 68 apps, whereas there are, in total (at the time of writing) 1,112 apps in the overall F-Droid community [7]. There could potentially be some sampling or other biases if we restrict ourselves solely to the benchmark apps. Furthermore, since SAPIENZ and the other techniques use randomised algorithms, it is widely regarded as best practice to perform an inferential statistical analysis of the performance of each algorithm, reporting statistical significance and effect size [35, 149]. Therefore, RQ4 investigates the findings that can be reported using statistical significance and effect size on multiple runs of the tools, each applied to a random sample of apps from the 1,112 F-Droid apps publicly available:

**RQ4 (Statistical significance and effect size):** How does SAPIENZ perform, compare to the state of the art and the state of practice, on randomly selected apps, with inferential statistical testing?

Finally, we want to investigate the usefulness of the SAPIENZ technique on real-world commercial apps. Therefore, we follow the practice adopted by some previous authors [142, 239] of applying the technique to a large number of popular apps in Google Play.

This avoids the potential bias of applying the technique only to apps chosen from F-Droid, which does not contain any of the most popular apps in current use. Since we do not have access to the source code of these popular commercial apps, it also tests the effectiveness of the technique when used in 'black box mode', where it has least available information to guide the test generation process, and only high level, non-invasive, 'skin coverage' instrumentation is possible.

**RQ5 (Usefulness):** Can SAPIENZ find any real bugs on popular closed-source real-world apps?

### 3.3.1 Experimental Setup

We conduct three studies to answer the above research questions: Study 1 addresses RQ0 to RQ3, Study 2 addresses RQ4 and Study 3 addresses RQ5. Study 1 and Study 2 are based on the execution of the testing approaches under evaluation on a single PC. Study 3 augments this, by using real-world physical (Samsung and Google) devices to demonstrate the practicality of SAPIENZ. For all these studies, we evaluate on Android KitKat version (API 19) because it is the most widely-used version [3] at the time of writing. All techniques under evaluation are fully automated. We choose not to provide manual assistance (e.g., logins) in testing the subjects, because we aim for an unbiased and rigorous assessment of what can be achieved entirely automatically.

Since Dynodroid itself manipulates the emulator and depends on its own customised Android system image, we follow its user guide [6] and use its own image file to execute the tool. For all the approaches under evaluation, we limit only the execution time and the assigned hardware resource, so that our comparison is direct head-to-head test effectiveness achieved in a certain amount of elapsed wall-clock time. This setting is consistent with the benchmark study conducted by Choudhary et al. [84], which allows us to perform a direct comparison with the results in that previous study.

We set SAPIENZ's crossover and mutation probability to 0.7 and 0.3 respectively. The maximum generation is set to 100 with the population size of 50 and each individual contains 5 test cases. None of the parameters available to SAPIENZ are tuned; all remain

set at the same value throughout all our experiments. We adopt this approach in order to ensure that the comparison is strictly fair; results for SAPIENZ might be improved by tuning, but this might also introduce bias and unfairness in the experimentation. We conducted Study 1 and Study 2 on a PC with a single hexa-core 3.50GHz CPU and 16GB RAM on Ubuntu 14.04. For Study 3, we also use a mobile device *Samsung Galaxy Note II* and a cluster of 10 *Google Nexus 7* (2013 version) tablets.

For Study 1, we test each subject for one hour by using each tool under evaluation. We record their achieved coverage every 5 minutes. When comparing fault-revealing test sequence lengths, we need to be careful to normalise the results: each technique might find a different number of faults, so measuring the total length of fault-revealing test sequences would be unfair. Rather, we compare the mean length of the fault-revealing test sequences returned by each approach. We count an atomic event as one event and decompose our high-level *motif genes* into multiple atomic events for a fair comparison.

For Study 2, we use random selection to identify 10 subjects from the 1,112 apps in the overall F-Droid set. We conduct an inferential statistical analysis of the performance of each of the Android testing techniques applied to these randomly selected apps. Details of the 10 randomly selected apps can be found in the left-hand columns of Table 3.4. Since we cannot rely on Gaussian (aka 'Normal') distribution of test results, we use a non-parametric multiple comparison inferential statistical significance test, the Kruskal-Wallis test [64] (at the 0.05 alpha level) with the Bonferroni correction, and the Vargha-Delaney effect size measure [369], as widely recommended [35, 149]. The differences between approaches are characterised as small, medium and large when the $\hat{A}_{12}$ effect size exceeds 0.56, 0.64, and 0.71, respectively. We repeat each experiment 20 times to provide a sample of runs for statistical analysis. In total, this more rigorous statistical evaluation requires 25 days of execution time.

Since Study 3 concerns the evaluation of SAPIENZ on 1,000 Google Play apps, the evaluation on 1,000 apps with real-world complexity is inherently time-consuming. Fortunately, since we are interested in the usefulness of the technique, we want to investigate whether it can find faults quickly. Therefore, we restrict the wall-clock execution time for this study to 30 minutes per app per setting. Furthermore, since emulators may

not reflect real device behaviour perfectly, we conduct this study under three device settings: on a PC with emulators, on a smart mobile device (*Samsung Note II*) and on a small cluster of 10 tablets (*Google Nexus 7*). The entire computation time of the experiment, on all 1,000 apps under three settings, to answer RQ5 is 1,050 hours (nearly 44 days); 500 hours on emulators, 500 hours on the Samsung Note II and 500/10 hours on the Google Nexus 7 tablets. In this study, we use only the non-invasive 'skin coverage' to guide SAPIENZ, so the results are a lower bound on the performance that would be observed by a developer, who could have access to source code and could therefore exploit the finer granularity levels of coverage.

### 3.3.2 Subject Dataset Collection

Our empirical evaluation Study 1 is conducted on the 68 benchmark F-Droid apps from the recent thorough study by Choudhary et al. [84]. Similarly, our Study 2 is based on 10 re-sampled OSS apps from the F-Droid community. Detailed information regarding these 10 apps is presented in Table 3.4.

Our dataset collected for Study 3 has 1,000 apps which consist of 500 most popular apps and 500 most popular new apps ranked by Google. Information about the distribution of categories and the range of downloads of the apps is shown in Figure 3.6 and Figure 3.7. The categories of the selected apps are wide and varied. This dataset was created from a snapshot of the Google Play store on August 16, 2015. For each app, we crawled the rating and ranking attributes along with its APK file.

### 3.3.3 State of the Art and Practice

According to the thorough empirical study by Choudhary et al. [84], existing techniques fail to outperform the standard Monkey Android testing tool in 'continuous mode'. In this mode, each testing tool is given one hour execution time and the same hardware configuration. We therefore chose to evaluate in the same way, comparing against Monkey and Dynodroid, which Choudhary et al. found to perform best among the research prototype techniques (beating recently proposed techniques including black

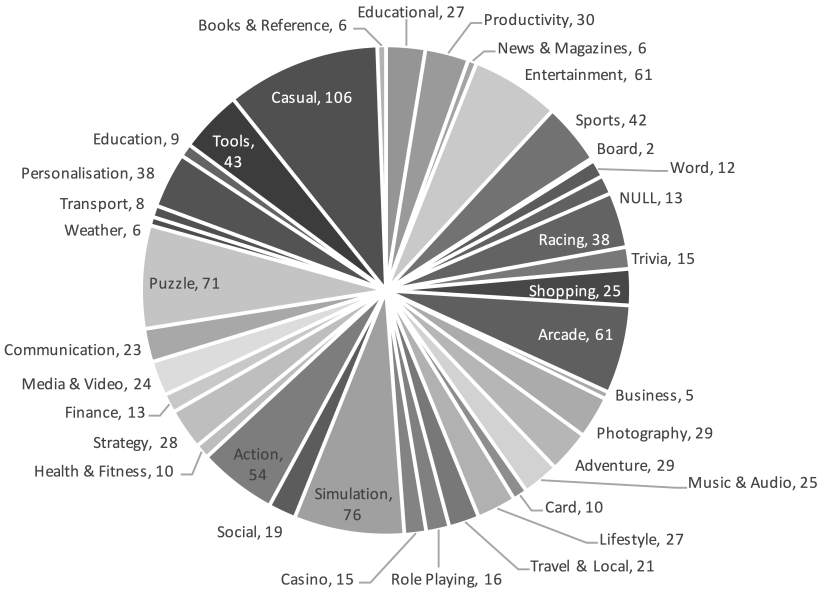



Figure 3.6: Category distribution of the 1,000 Google Play apps

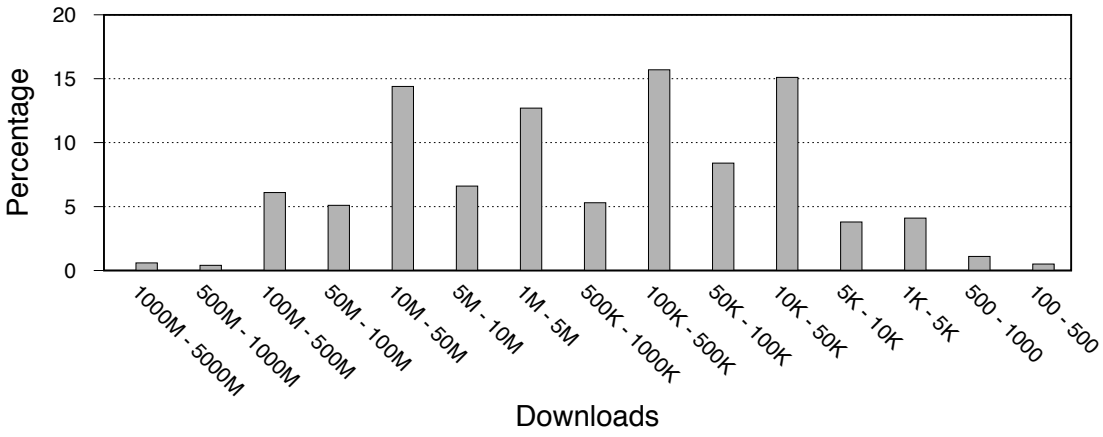



Figure 3.7: Download distribution of the 1,000 Google Play apps

box based AndroidRipper [28], A$^3$E [41], PUMA [142] and white-box based ACTEve [30]). Monkey and Dynodroid also performed best in a slightly more recent study [273], and, therefore, if Sapienz outperforms both Monkey and Dynodroid, we will also have reasonable evidence to conclude that it is likely to outperform AndroidRipper [28], A$^3$E [41], PUMA [142] and ACTEve [30]. Note that Sapienz also yields a Pareto front at the end of its execution, which might be a useful by-product. However, we choose to evaluate Sapienz only in the 'continuous mode', for a fair comparison with Monkey and Dynodroid, which do not yield Pareto fronts.

### 3.3.4 Results

#### Study 1: Benchmark Subjects

The detailed experimental results on each subject for Study 1 are given in Table 3.3, where 'Coverage' reports statement coverage achieved by each of the three tools, '#Crashes' indicates the number of unique crashes detected by each and 'Length' reports the fault-revealing test sequence length for each. The column headings 'M', 'D' and 'S' refer to the three tools we compare; Monkey, Dynodroid and SAPIENZ. The entry '*' indicates the tool cannot start the corresponding app, while the entry '-' indicates that the fault-revealing length is undefined, because no faults were found.

**RQ0 (Experimental replication).** We first evaluate Monkey and Dynodroid to check that our experiment infrastructure replicates the results reported by Choudhary et al. [84]. We calculated progressive average coverages across all 68 subjects every 5 minutes for each of the three techniques and report the direct comparison on the final coverages achieved. The progressive coverages of Monkey and Dynodroid are shown in Figure 3.8. The shape of the growth in coverage over time very closely resembles the results reported by Choudhary et al. [84]. However, the final coverage values achieved by these two tools are slightly higher than those reported by Choudhary et al. This may be caused by the hardware setting: Choudhary et al. ran the experiments on virtual machines while we conducted our experiments on a physical PC which may be faster. Since the overall growth trend closely resembles the results of Choudhary et al., and given that better performance only raises the bar that SAPIENZ must clear in order to outperform them, we believe these results indicate we have a firm foundation on which to perform our subsequent experiments.

**RQ1 (Code coverage).** We used an identical evaluation approach for SAPIENZ as that used in the replication study reported in RQ0 for Monkey and Dynodroid. As can be seen from Figure 3.8, SAPIENZ outperformed Monkey and Dynodroid from the $10^{\text{th}}$ minute onwards, finally achieving the highest overall statement coverage at the end of the hour's experimental time allowed for each of the 68 subjects. To further investigate these results, Figure 3.9 presents the boxplots (for which a circle indicates

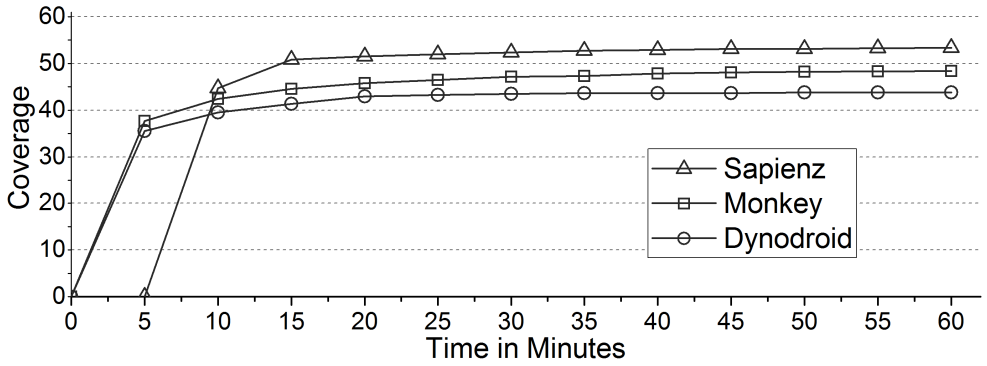



Figure 3.8: Progressive coverage on benchmark apps

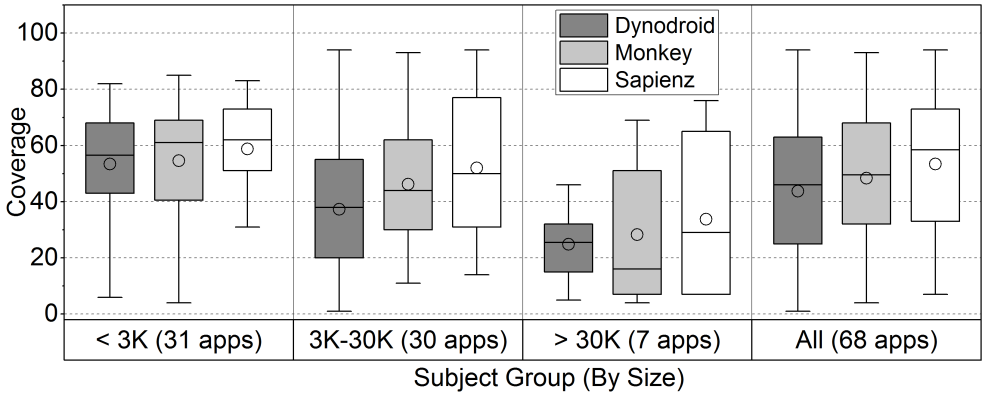



Figure 3.9: Code coverage on the 68 benchmark apps

the mean) of the final coverage results for apps grouped by size-of-app. This analysis reveals that SAPIENZ achieved the highest mean coverage across all four app size groups. We conclude that there is evidence from the 68 benchmark apps that SAPIENZ can attain and maintain superior coverage after approximately 10 minutes of execution on a standard equipment.

**RQ2 (Fault revelation).** In answering RQ2, we report not only on the number of crashes found by each technique, but also the overlap between the crashes found by each technique. This allows us to investigate whether the techniques are complementary, or whether one subsumes another, as well as reporting on the overall effectiveness (in terms of number of crashes found). Of course a crash may be triggered by different test sequences, so we report *unique* crashes, considering a crash to be unique when its stack trace differs from all others. We excluded those crashes caused by the Android system or the test harness itself, which were not caused by the faults from the subjects. Such crashes can be identified by checking the corresponding stack traces. A recent study

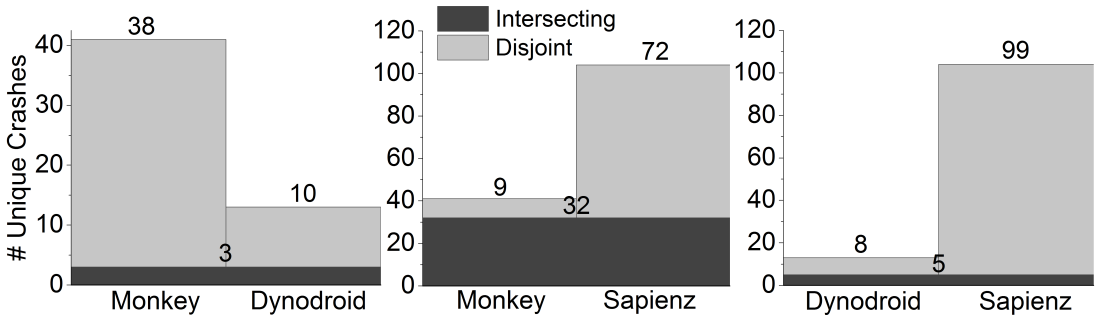



Figure 3.10: Pairwise comparison on found crashes

Table 3.1: Statistics on found crashes

| App Crashes | Monkey | Dynodroid | Sapienz |
|---|---|---|---|
| # App Crashed | 24 | 13 | 41 |
| # Unique Crashes | 41 | 13 | 104 |
| # Total Crashes | 1,196 | 125 | 6,866 |

[273] has highlighted this issue and pointed out that these crashes are, essentially false positives, so should not be counted.

As shown in Table 3.1, SAPIENZ revealed the largest number of both unique and total crashes in 41 of the 68 apps. SAPIENZ also found 30 unique crashes in 14 apps for which neither Monkey nor Dynodroid found any crash. We also provide a pairwise comparison of the unique crashes found in Figure 3.10 (where the black bars show common crashes; those revealed by both techniques): Across the 68 subjects, SAPIENZ found 72 and 99 unique crashes, undetected by Monkey and Dynodroid respectively, while it missed only 9 crashes found by Monkey and 8 by Dynodroid. We conclude that there is strong evidence from the 68 benchmark apps that SAPIENZ outperforms both Monkey and Dynodroid in terms of fault revelation, as measured by the number of crashes found.

**RQ3 (Sequence length).** Table 3.2 shows the mean length of fault-revealing test sequences of the three tools, grouped by various subject size ranges (where the group sizes are given in the brackets). On all subject groups except '3K-30K', SAPIENZ generated the shortest fault-revealing test sequences. On the '3K-30K' subject group, Dynodroid generated the shortest fault-revealing test sequences (although its code coverage and number of found crashes are lower than SAPIENZ). We conclude that there is strong evidence from the 68 benchmark apps that SAPIENZ outperforms the fault-revealing test

Table 3.2: Fault-revealing test sequence length

| | | Monkey | Dynodroid | Sapienz |
|---|---|---|---|---|
| Size | < 3K (31) | 13,843 | 186 | 132 |
| | 3K-30K (30) | 14,775 | 77 | 153 |
| | > 30K ( 7) | 21,501 | 276 | 169 |
| Overall (68) | | 15,305 | 161 | 149 |

sequence length of Monkey, and that on larger subjects it also outperforms Dynodroid.

### Study 2: Inferential Statistical Analysis

**RQ4 (Statistical significance and effect size).** For all 10 randomly sampled F-Droid programs, and for all three criteria of interest, SAPIENZ outperformed both Monkey and Dynodroid. However, in this study, we are concerned with the statistical significance in effect size of these results. We first present the boxplots of the performance comparison on 10 F-Droid subjects, as shown in Figure 3.11.

Table 3.4 shows Vargha-Delaney $\hat{A}_{12}$ effect size for the three objectives, coverage, the number of crashes found and fault-revealing sequence length. For each objective, the columns contain the effect size comparisons for SAPIENZ-Monkey (S-M), SAPIENZ-Dynodroid (S-D), and, for completeness, Monkey-Dynodroid (M-D), where the result is significant. As shown in the table, SAPIENZ significantly outperforms Monkey with large effect size on 7/10 subjects for coverage, 8/10 for crashes, and 10/10 for length (with large effect size). SAPIENZ significantly outperforms Dynodroid, with large effect size on 9/10 subjects for coverage, 9/10 for crashes and 10/10 for length. We also replicated the finding of Choudhary et al. [84] that Monkey tends to outperform Dynodroid, but further note that it does so less conclusively than SAPIENZ does. The overall results suggest that SAPIENZ outperforms both the state-of-the-art and the state-of-practice approaches on all three objectives.

### Study 3: Top 1,000 Popular Apps

**RQ5 (Usefulness).** In total, SAPIENZ found 558 unique crashes in 329 of the 1,000 Google Play apps to which it was applied. In the previous study of Dynodroid [239], the authors also tested top 1,000 apps, however the budget used and total number of found

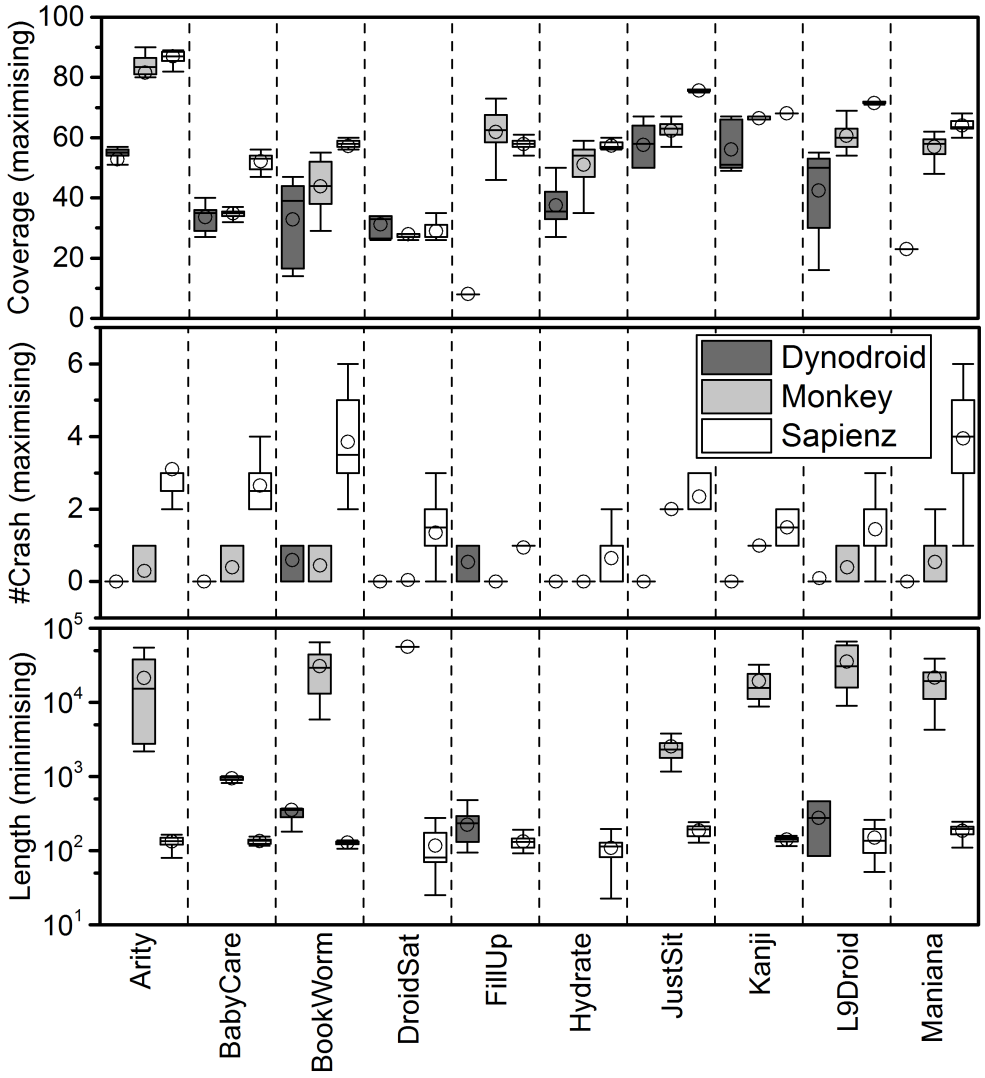



Figure 3.11: Performance comparison on 10 F-Droid subjects (boxplots grouped by subject)

unique crashes are not mentioned. The authors found 6 bugs (that led to non-native crashes) in 5 out of 1,000 apps tested. Our 558 unique crashes were caused by 22 types of errors/exceptions, where 161 are 'native' crashes, indicating that the crash occurred outside the Android Java Virtual Machine, while executing the app's native code. The top three most common non-native crashes are due to null pointers, missing activities and out of memory.

We reported the non-native crashes to the app provider, giving a stack trace for each crash type. An example crash stack trace found by SAPIENZ is listed in Figure 3.12. In total, we reported 175 crashes[3]. Unfortunately, since these apps are commercial

[3]For each app, we reported the first found crash that corresponds to each non-native crash type.

```
1 // CRASH: com.*
2 // Short Msg: java.lang.ArrayIndexOutOfBoundsException <- Crash Type
3 // Long Msg: java.lang.ArrayIndexOutOfBoundsException: rowIndex is less than 0.
4 // java.lang.ArrayIndexOutOfBoundsException: rowIndex is less than 0.
5 // at io.realm.internal.TableView.nativeGetSourceRowIndex(Native Method) <- Crash
      Location
6 // at io.realm.internal.TableView.getSourceRowIndex(TableView.java:161)
7 // at io.realm.RealmResults.get(RealmResults.java:114)
8 // at io.realm.RealmBaseAdapter.getItem(RealmBaseAdapter.java:73)
9 ...
```

Figure 3.12: An example of crash stack traces found by Sapienz

apps, we do not have direct access to the developers, as one might in an open-source environment, but we were able to contact only the associated customer support team. We got 58 replies in total, excluding those that were automatic generated. For such a 'cold call' outreach activity, 58 from 175 emails is relatively high [122, 189].

Of these 58 replies, in 27 cases we got feedback from the app developers (after our email was redirected by their customer support teams). Furthermore, 14 developer teams confirmed that the crashes resulted from real faults in their apps, and 6 of them have already fixed the reported crashes. Among the 13 unconfirmed crashes out of 27 developer replies, 6 indicated that our reports were helpful or that the developers were working on the issue. A further 6 respondents seek additional information. One of the 13 responded that they could not identify the cause of the crash.

We list the anonymised details[4] of these 14 faults confirmed by developers in Table 3.5: These 14 apps vary greatly in categories and install numbers, with at least 148 million installs in total. The 6 confirmed faults, with further fixes from their developers are labelled as 'Confirmed' in the 'Fixed' column. For the remaining 8 apps, we found that 7 of the confirmed crashes can no longer be observed when testing their most recent versions. However since we have not received confirmation from developers that these faults are definitely fixed, we label them as 'Unconfirmed' in the 'Fixed' column. We observed only one of the confirmed faults was not fixed (still crashes).

We did not report native crashes because their stack traces do not explicitly point to the source lines of the potential faults.

[4]App versions are omitted for anonymity.

**An Analysis on Sapienz' Found Crashes**

**Prevalence of app crashes.** SAPIENZ in total found 558 crashes in the 324 out of 1,000 Android apps. In all categories, no matter how small the sample, SAPIENZ did found crashed apps. Table 3.6 shows the number and proportions of the crashed apps and the number of crashes in each category. From this figure we can see there are 32% of the apps that crash, on average, in each category. We immediately notice that game apps have a greater number of crashes than other categories. The reason is that the game category has many more apps than other categories. We also observe that the quality of crashing apps varies from category to category: Over 50% of apps in the Transport, Finance and Photography categories were crashed at least once, while only 20% apps found in the Health & Fitness, Personalisation and Productivity categories crashed.

Table 3.7 shows the number and proportions of each type of crash, extracted from the stack traces. Among all crashes, 28% only produce memory or native error information (without any stack traces). Therefore, we cannot determine their type of crash. From the table, we can see there are 21 known types of crash, and more than 40% of them are either of the NullPointer (26.7%) or the ActivityNotFound(19.7%) type. We also observe a 'long tail' for which the occurrence (of 13 crash types) is very rare; each is lower than 1%.

Although many types of crashes, such as, NullPointer, OutOfMemory and ArrayIndex-OutOfBounds, are also prevalent in traditional software systems, we also observe some app-specific crashing behaviours. For example, ActivityNotFound, IllegalState and WindowManagerBadToken. Figure 3.13 shows a tag cloud generated from all crashing stack traces. Once again, this reveals some app-specific topics, such as 'activity' and 'view'. The design of future app testing techniques should perhaps leverage some of these app-specific behaviours to test apps more effectively and efficiently.

**Locations of app crashes.** Figure 3.15 shows the number of crashes found at various locations including developer's implementation, third party library, Android native code and Android OS. It is obvious that the majority of the crashes were found in the


Figure 3.13: Word cloud on crash stack trace content

developer code. However, there are also a number of interesting observations about other locations. For example, we found 173 crashes are located in the third party library code (38%). Table 3.8 lists the top 5 libraries that have the largest number of crashes. These crashes are harder to debug, because the developers may not have the source code for many third party libraries. We took the GDX library as an example and looked at its Github repository. We observed that some crashes found in our study were also reported in the issue page in the git repository. To avoid this type of crash, more testing should spend on checking the integration between the library and app code.

There are 40 crashes located in the Android platform. These crashes are even harder to debug because the generated stack traces do not explicit point to the potential problematic statements in the subject. We found 161 Android native crashes which are located at the native JNI code or Android native platform code. Table 3.14 shows an example of the Android native crashes. These crashes do not output human readable stack traces, thus it is not straightforward to debug them. We also found 39 crashes occurred in the Android UI systems, which happened when Android OS is loading the app and initializing the GUI, but before entering the actual app logic code is executed. We manually investigated these crashes and found most of them belong to the IllegalArgument type of crash. They often only occur with certain mobile devices according to the Android app issues reported on Github. This result provides further evidence that the

```
1  // CRASH: com.ea.game.simcitymobile_row
2  // Short Msg: Native crash
3  // Long Msg: Native crash: Segmentation fault
4  // *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
5  // Revision: '0'
6  // pid: 25731, tid: 25749, name: Thread-1866 >>> com.ea.game.simcitymobile_row <<<
7  // signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000
8  // r0 00000000 r1 76f34207 r2 00000000 r3 00000000
9  // r4 00000000 r5 76f6bb18 r6 00000200 r7 76f6d8c4
10 // r8 76342e10 r9 00000000 sl 76f3b1d8 fp 00000000
11 // ip 00000000 sp 76342c4c lr 76e63f30 pc 76ecabf0 cpsr 20070010
12 ...
```

Figure 3.14: An example of native crash stack traces found by Sapienz

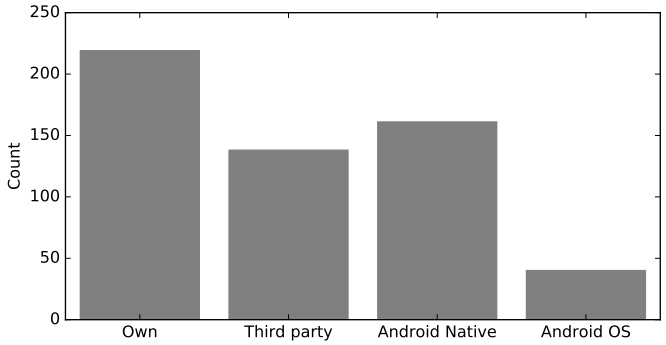



Figure 3.15: Crash locations

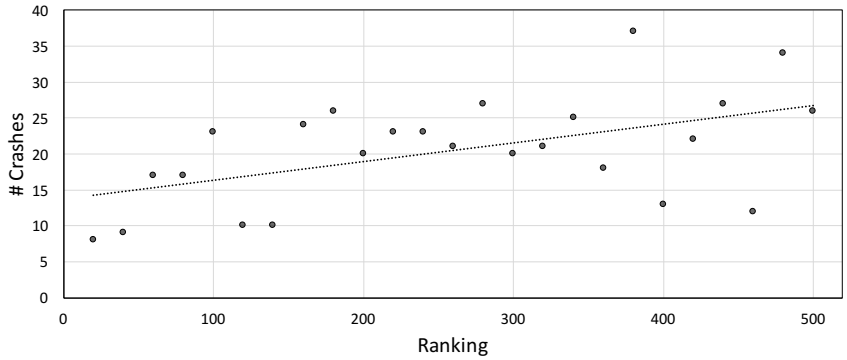
Android fragment is a critical problem, and testers should also consider testing their apps on various device models.

**Relationship of crashes and ratings and rankings.** We applied Pearson correlation analysis to check the relationships between crashes in an app with its popularity and user satisfaction level, as shown in Figure 3.16. We found a moderate correlation between the number of crashes and the ranking ($\rho = 0.51$). This was expected because the popular apps might be expected to be more reliable. We also expected a similar correlation for ratings. However, there was no such correlation ($\rho = -0.17$).

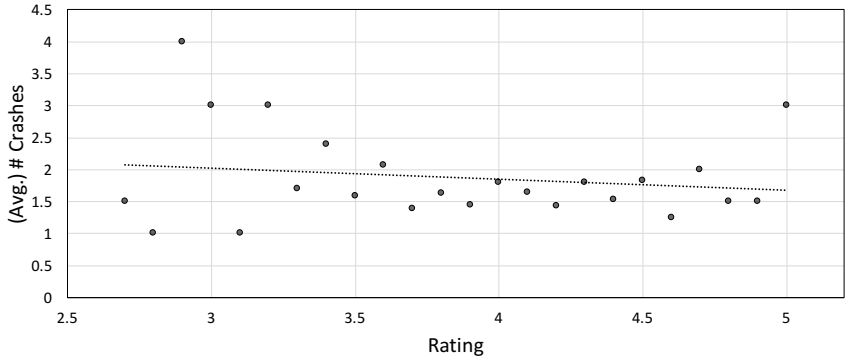
### 3.3.5 Threats to Validity

Like any empirical study, there are potential threats to validity of our experimental results:

**Internal validity**: Threats to internal validity concern factors in our experimental methodology that may affect our results. For Study 1, 50 of the 68 ASE benchmark




(a) Crash-ranking correlation.




(b) Crash-rating correlation.

Figure 3.16: Correlation between the number of crashes and rating and ranking

subjects originate in a single article [239], which might have resulted in selection bias. To mitigate this issue, we conducted Study 2 on 10 open-source apps, selected using unbiased random sampling. Regarding the particular SAPIENZ implementation, we implemented only a single motif pattern to exercise all text fields and clickable UI widgets under the corresponding view, which is applicable to all apps. The performance of SAPIENZ may improve when considering different motif patterns, but could not be worse, since this single option will always be available. Also, the choice of parameter setting for each of the three tools may affect their performance significantly. To reduce this threat, we followed the default configurations for Monkey and Dynodroid, as used in the previous thorough benchmark assessment study Choudhary et al. [84] and we resisted any temptation to tune SAPIENZ.

**External validity**: Threats to external validity arise when the experimental results cannot be generalised. Like all empirical studies, we are limited in the number of subject systems to which we can apply our tools and techniques. Our results will

not necessarily generalise beyond the 1,078 apps to which we have applied SAPIENZ. However, we think it promising that the technique applies, out of the box, to so many different apps, none of which have been 'cherry picked' (nor in any other way 'chosen' by the experimenters themselves). It is possible, of course, that the 1,000 most popular apps, and the F-Droid open-source apps, have peculiar characteristics not shared by other classes of apps, for which the performance of the three techniques we studied in this chapter may differ. We also only evaluated our approach on a single version of the Android platform. Although the most widely-used version, the rapid evolution of the Android system, means that the performance of three evaluated techniques may vary as subsequent versions become available.

## 3.4 Summary

This chapter introduced the multi-objective search-based software testing technique SAPIENZ for automated Android app testing. SAPIENZ supports multi-level instrumentation and remains applicable, even when only the app's APK file (and nothing else) is available. Its evolutionary algorithm continuously optimises for coverage, sequence length and the number of crashes found, seeking to reveal as many crashes as possible, while minimising the required operations.

Our evaluation results on open-source apps have shown that SAPIENZ outperforms the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, on all three objectives for almost all the subjects. The only exception is the relatively small (3K-30K lines of code) F-Droid open-source apps in the benchmark suite, for which Dynodroid produced shorter fault-revealing test sequences, although it achieved less coverage and revealed fewer crashes.

We also believe that there is compelling evidence that SAPIENZ is a practical and useful testing tool, since it was able to find 558 unique crashes in the top 1,000 most popular Android apps, 14 of which have already been confirmed as caused by real faults.

Table 3.3: Results on the 68 benchmark apps

| Subject | Coverage | | | #Crashes | | | Length | | |
|---|---|---|---|---|---|---|---|---|---|
| | **M** | **D** | **S** | **M** | **D** | **S** | **M** | **D** | **S** |
| a2dp | 43 | 29 | 46 | 0 | 1 | 3 | - | 315 | 148 |
| aarddict | 14 | 46 | 18 | 0 | 0 | 0 | - | - | - |
| aLogCat | 68 | 49 | 71 | 0 | 0 | 2 | - | - | 114 |
| Amazed | 66 | 63 | 69 | 1 | 0 | 1 | 1429 | - | 96 |
| AnyCut | 63 | 65 | 66 | 0 | 0 | 1 | - | - | 103 |
| baterrydog | 64 | 66 | 67 | 0 | 1 | 1 | - | 81 | 173 |
| swiftp | 13 | 13 | 14 | 0 | 0 | 0 | - | - | - |
| Book-Catalogue | 46 | 27 | 33 | 1 | 0 | 1 | 1941 | - | 177 |
| bites | 38 | 25 | 41 | 1 | 0 | 1 | 19124 | - | 116 |
| battery | 76 | 68 | 79 | 0 | 0 | 4 | - | - | 198 |
| addi | 16 | 26 | 20 | 2 | 1 | 2 | 1367 | 315 | 129 |
| alarmclock | 72 | 51 | 77 | 4 | 1 | 5 | 1716 | 170 | 144 |
| manpages | 64 | 68 | 75 | 0 | 0 | 3 | - | - | 120 |
| mileage | 40 | 25 | 54 | 2 | 1 | 4 | 878 | 390 | 153 |
| autoanswer | 13 | 24 | 16 | 0 | 0 | 0 | - | - | - |
| hndroid | 4 | 6 | 10 | 2 | 1 | 2 | 206 | - | 117 |
| multismssender | 43 | 49 | 61 | 0 | 0 | 0 | - | - | - |
| worldclock | 93 | 94 | 94 | 0 | 0 | 1 | - | - | 98 |
| Nectroid | 69 | 46 | 76 | 1 | 0 | 2 | 416 | - | 118 |
| acal | 15 | 15 | 29 | 1 | 0 | 5 | 62717 | - | 177 |
| jamendo | 62 | 3 | 72 | 0 | 0 | 2 | - | - | 191 |
| aka | 79 | 76 | 84 | 1 | 0 | 7 | 42804 | - | 136 |
| yahtzee | 62 | 51 | 58 | 2 | 0 | 0 | 31767 | - | - |
| aagtl | 30 | 29 | 31 | 4 | 0 | 5 | 1756 | - | 188 |
| CountdownTimer | 60 | 62 | 62 | 0 | 0 | 0 | - | - | - |
| sanity | 32 | 1 | 19 | 2 | 1 | 2 | 8377 | 12 | 90 |
| dalvik-explorer | 69 | * | 73 | 2 | * | 4 | 3720 | * | 165 |
| Mirrored | 69 | 68 | 64 | 0 | 0 | 1 | - | - | 147 |
| dialer2 | 38 | 55 | 42 | 0 | 0 | 0 | - | - | - |
| DivideAndConquer | 85 | 72 | 83 | 0 | 0 | 2 | - | - | 186 |
| fileexplorer | 40 | 56 | 50 | 0 | 0 | 0 | - | - | - |
| gestures | 36 | 48 | 52 | 0 | 0 | 0 | - | - | - |
| hotdeath | 78 | 3 | 79 | 1 | 0 | 3 | 63975 | - | 152 |
| adsdroid | 23 | 36 | 38 | 2 | 1 | 1 | 356 | 48 | 128 |
| myLock | 28 | 33 | 31 | 0 | 0 | 0 | - | - | - |
| lockpatterngenerator | 78 | 79 | 81 | 0 | 0 | 0 | - | - | - |
| mnv | 49 | * | 67 | 2 | * | 4 | 30381 | * | 150 |
| aGrep | * | 38 | * | * | 0 | * | * | - | * |
| k9mail | 7 | 5 | 7 | 0 | 0 | 1 | - | - | 238 |
| LolcatBuilder | 24 | 23 | 31 | 0 | 0 | 0 | - | - | - |
| MunchLife | 70 | 73 | 76 | 0 | 0 | 0 | - | - | - |
| MyExpenses | 51 | 25 | 65 | 0 | 1 | 2 | - | 67 | 150 |
| LNM | 58 | 66 | 60 | 1 | 0 | 1 | 51621 | - | 48 |
| netcounter | 44 | 63 | 77 | 0 | 0 | 2 | - | - | 156 |
| bomber | 76 | 70 | 73 | 0 | 0 | 0 | - | - | - |
| frozenbubble | * | 63 | * | * | 0 | * | * | - | * |
| fantastischmemo | 36 | 9 | 60 | 1 | 0 | 6 | 25375 | - | 156 |
| blokish | 50 | 50 | 52 | 1 | 1 | 2 | 2512 | 252 | 194 |
| zooborns | 35 | 38 | 36 | 0 | 0 | 0 | - | - | - |
| importcontacts | 41 | 43 | 42 | 0 | 0 | 0 | - | - | - |
| wikipedia | 36 | 32 | 32 | 0 | 0 | 5 | - | - | 232 |
| PasswordMaker | 63 | 53 | 64 | 3 | 0 | 1 | 3406 | - | 180 |
| passwordmanager | 11 | 7 | 16 | 0 | 0 | 0 | - | - | - |
| Photostream | 16 | 23 | 38 | 1 | 1 | 2 | 317 | 29 | 125 |
| QuickSettings | 50 | 33 | 50 | 0 | 0 | 1 | - | - | 134 |
| RandomMusicPlayer | 58 | 82 | 59 | 0 | 0 | 0 | - | - | - |
| Ringdroid | 26 | * | 29 | 1 | * | 2 | 550 | * | 161 |
| soundboard | 42 | 60 | 53 | 0 | 0 | 0 | - | - | - |
| SpriteMethodTest | 82 | 37 | 83 | 0 | 0 | 0 | - | - | - |
| SpriteText | 59 | 57 | 62 | 0 | 0 | 0 | - | - | - |
| SyncMyPix | 21 | 20 | 22 | 0 | 0 | 4 | - | - | 187 |
| tippy | 83 | 48 | 83 | 0 | 0 | 0 | - | - | - |
| tomdroid | 55 | 43 | 58 | 0 | 1 | 1 | - | 165 | 91 |
| Translate | 48 | 45 | 49 | 0 | 0 | 0 | - | - | - |
| Triangle | 76 | 69 | 79 | 0 | 0 | 0 | - | - | - |
| weight-chart | 58 | 57 | 77 | 2 | 1 | 4 | 10588 | 236 | 186 |
| whohasmystuff | 74 | * | 80 | 0 | * | 0 | - | * | - |
| Wordpress | 4 | * | 7 | 0 | * | 1 | - | * | 137 |

Table 3.4: Vargha-Delaney effect size ('-' indicates a statistically insignificant result)

| Subject | Description | Ver. | Date | SLOC | Coverage | | | #Crash | | | Length | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | S-M | S-D | M-D | S-M | S-D | M-D | S-M | S-D | M-D |
| Arity | Scientific calculator | 1.27 | 2012-02-11 | 2,821 | - | 1.00 | 1.00 | - | 1.00 | 0.98 | 1.00 | 1.00 | - |
| BabyCare | Timer for when to feed baby | 1.5 | 2012-08-23 | 8,561 | 1.00 | 1.00 | - | 0.84 | 0.92 | - | 1.00 | 1.00 | - |
| BookWorm | Book collection manager | 1.0.18 | 2011-05-04 | 7,589 | 0.96 | 1.00 | - | 0.97 | 1.00 | - | 1.00 | 0.95 | - |
| DroidSat | Satellite viewer | 2.52 | 2015-01-11 | 15,149 | - | - | - | 1.00 | 1.00 | - | 0.90 | 0.90 | - |
| FillUp | Calculate fuel mileage | 1.7.2 | 2015-03-10 | 10,400 | - | 1.00 | 1.00 | 0.73 | 0.73 | - | 0.95 | 0.80 | 0.23 |
| Hydrate | Set targets for water intake | 1.5 | 2013-12-09 | 2,728 | 0.85 | 1.00 | 0.92 | 0.95 | - | 0.23 | 0.73 | 0.73 | - |
| JustSit | Meditation timer | 0.3.3 | 2012-07-26 | 728 | 1.00 | 1.00 | - | 1.00 | 1.00 | - | 1.00 | 1.00 | 1.00 |
| Kanji | Character recognition | 1.0 | 2012-10-30 | 200,154 | 1.00 | 1.00 | 0.84 | - | 1.00 | 1.00 | 1.00 | 1.00 | 0.98 |
| L9Droid | Interactive fiction | 0.6 | 2015-01-06 | 18,040 | 1.00 | 1.00 | 0.99 | 0.89 | 0.90 | - | 0.94 | 0.91 | - |
| Maniana | User-friendly todo list | 1.26 | 2013-06-28 | 20,263 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | - | 1.00 | 1.00 | - |

Table 3.5: Confirmed app faults identified by Sapienz

| App | Category | Installs | Caused By | Device | Description | Fixed |
|---|---|---|---|---|---|---|
| P* | Photography | 10M-50M | `NullPointer` | Nexus 7 | Unable to start activity from a customer support SDK. | Unconfirmed |
| K* | Simulation | 10M-50M | `NullPointer` | Nexus 7 | Concurrent error while executing `doInBackground()` | Unconfirmed |
| B* | Business | 10K-50K | `NullPointer` | Nexus 7 | Null object reference in a third party SDK | No |
| D* | Education | 500K-1M | `NullPointer` | Emulator | Exception from event handler `onOptionsItemSelected()` | Confirmed |
| T* | Simulation | 10K-50K | `NullPointer` | Emulator | Exception from `onAnimationEnd()` in `FlipGameActivity` | Confirmed |
| T* | Lifestyle | 500K-1M | `NullPointer` | Emulator | Error when `CameraUpdateFactory` is not initialized | Confirmed |
| T* | Transport | 1M-5M | `NullPointer` | Emulator | Exception from `onClick()` in `StationInfoFragment` | Confirmed |
| S* | Education | 1M-5M | `NullPointer` | Emulator | Unable to start a third party activity | Unconfirmed |
| T* | Weather | 10M-50M | `NullPointer` | Emulator | Error when `CameraUpdateFactory` is not initialized | Unconfirmed |
| W* | Weather | 10K-50K | `OutOfMemory` | Note II | Error inflating class on binary XML file | Unconfirmed |
| S* | Puzzle | 5M-10M | `ActivityNotFound` | Note II | No Activity found to handle `SHARE_GOOGLE` Intent. | Unconfirmed |
| F* | Photography | 10M-50M | `NullPointer` | Note II | Exception from `onGlobalLayout()` in `ViewUtil` | Confirmed |
| T* | Music&Audio | 100M-500M | `NullPointer` | Note II | Unable to start the activity of `PlayerActivity` | Unconfirmed |
| P* | Music&Audio | 5K-10K | `ActivityNotFound` | Note II | No Activity found to handle a `View` Intent | Confirmed |

Table 3.6: Crashed Apps by Category

| Category | # of Crashed Apps | % of Crashed Apps | # of Crashes |
|---|---|---|---|
| Game | 176 | 32% | 274 |
| Entertainment | 17 | 29% | 30 |
| Photography | 14 | 52% | 33 |
| Tools | 14 | 33% | 26 |
| Media & Video | 12 | 48% | 25 |
| Sports | 11 | 28% | 20 |
| Personalisation | 8 | 22% | 15 |
| Finance | 7 | 58% | 15 |
| Music & Audio | 7 | 28% | 12 |
| Lifestyle | 7 | 26% | 10 |
| Shopping | 7 | 28% | 10 |
| Transport | 6 | 75% | 13 |
| Travel & Local | 6 | 30% | 12 |
| Productivity | 6 | 21% | 9 |
| Social | 6 | 33% | 7 |
| Communication | 5 | 23% | 6 |
| Education | 4 | 44% | 19 |
| News & Magazines | 3 | 50% | 9 |
| Health & Fitness | 2 | 22% | 5 |
| Business | 2 | 50% | 3 |
| Weather | 2 | 40% | 3 |
| Books & Reference | 2 | 33% | 2 |
| Total | 324 | 32% | 558 |

Table 3.7: Crash Type Distribution

| Crash Type | # of Crash | Percentage |
|---|---|---|
| NullPointer | 149 | 26.70% |
| ActivityNotFound | 110 | 19.71% |
| OutOfMemory | 37 | 6.63% |
| IllegalState | 29 | 5.20% |
| IllegalArgument | 15 | 2.69% |
| RuntimeException | 13 | 2.33% |
| ArrayIndexOutOfBounds | 9 | 1.61% |
| IndexOutOfBounds | 7 | 1.25% |
| ConcurrentModification | 5 | 0.90% |
| WindowManagerBadToken | 5 | 0.90% |
| UnsatisfiedLinkError | 3 | 0.54% |
| NoClassDefFoundError | 3 | 0.54% |
| StackOverflowError | 2 | 0.36% |
| TransactionTooLarge | 2 | 0.36% |
| ErrnoException | 2 | 0.36% |
| UnsupportedOperation | 1 | 0.18% |
| ViewRootImplCalledFromWrongThread | 1 | 0.18% |
| TimeoutException | 1 | 0.18% |
| ResourcesNotFound | 1 | 0.18% |
| NetworkOnMainThread | 1 | 0.18% |
| NoSuchMethodError | 1 | 0.18% |
| Unknown | 161 | 28.85% |
| Total | 558 | 28.85% |

Table 3.8: Top 5 crashed third party library

| Name | Description |
|---|---|
| org.cocos2d | 2D Game Engine |
| com.badlogic.gdx | Cross-platform Game Framework |
| om.uservoice.uservoicesdk | In-app Customer Service |
| io.realm | Mobile Database |
| com.supersonicads | Mobile Advertising framework |

# Chapter 4

# Octopuz: Multi-objective Automated JavaScript Testing

In this chapter, we report on Octopuz, a conceptual replication of our previous Sapienz approach. We design the Octopuz system to use the Sapienz' multi-objective search technique for automated JavaScript testing, aiming to investigate whether the system replicates Sapienz' success at optimising three competing objectives (coverage, fault revelation, and test sequence length) during test evolution. The experimental results on 10 real-world JavaScript web applications show that Octopuz revealed 8 previously unknown, unique faults on 10 subjects, significantly outperforming the state-of-practice tool Gremlins (with large effect size), on 8 out of 10 subjects for coverage and 10 out of 10 for test sequence length. It also significantly outperformed Gremlins (with large effect size) for 2 of the 3 subjects for which either found a fault. In other experiments, Octopuz also outperformed the state-of-the-art techniques Artemis and JSeft with a mean coverage of 69%, revealing 4 real faults on the subjects (one of which was not found by either of the other two tools). This provides compelling evidence that the Sapienz approach for multi-objective search based testing is applicable to JavaScript test data generation.

## 4.1 Introduction

JavaScript is an increasingly important programming language for both websites and hybrid mobile apps. W[3]Techs reports that over 93% of all websites use JavaScript[1], while Gartner estimated that over 50% of mobile apps would be a hybrid of native code and web applications by 2016 [126]. JavaScript dominates the repositories on GitHub[2]. Popular mobile app frameworks such as Ionic also make significant use of JavaScript (Ionic's repository is 77.5% in JavaScript[3]), which enables the creation of hybrid mobile apps in JavaScript. The importance of websites and mobile apps for business, commerce, governmental transactions, and the technical and economic infrastructures [195, 216, 305] highlights the need for tools that can find bugs in JavaScript. However its event-driven and asynchronous features make it challenging to determine all possible (or likely) runtime application contexts. This increases the importance of automated testing techniques that can detect faults that might otherwise be missed by developers, eventually leading to software failures exposing to severe vulnerabilities.

Much existing work on test automation has focused on more traditional programming languages, targeted at desktop applications [70, 152]. While there have been automated test data generation techniques for web applications, these often cover server-side code, targeting web services implemented via languages such as PHP, rather than client-side code in JavaScript [26, 37, 47, 139, 252, 293]. These techniques are not suitable for testing client-side web applications from UI level as complex interaction events are required in order to cover the state space of the applications.

The existing state of practice for automated JavaScript test data generation relies on random test data generation. The most popular tool available on GitHub for fully automated JavaScript testing is Gremlins [251]. It has over 6,660 stars/likes and 284 forks on GitHub (at the time of writing), which provides random testing for JavaScript. However, random test data generation is known to be suboptimal, compared to more intelligent computational search techniques [152], and is widely regarded as merely a baseline sanity check against which such more intelligent techniques should be compared

[1] W[3]Techs survey: http://w3techs.com/technologies/details/cp-javascript/all/all
[2] Statistics on the programming language usage in GitHub: http://githut.info
[3] http://github.com/driftyco/ionic

[35, 149].

There are two more intelligent test data generation techniques for automated testing of JavaScript, that represent the current state of the art. Artemis [38], a JavaScript testing tool based on feedback-directed testing, was introduced by Artzi et al. in 2011, and remains an active project on GitHub[4], spanning over 4 years of development, and with over 1,458 commits. Feedback from the system's behaviour on test inputs generated, is used to guide the test data generation approach, with the goal of obtaining higher coverage (than would be obtained by a purely random approach). Artemis was evaluated on 10 JavaScript applications, ranging in size from 156 to 2,037 lines of code. More recently, the mutation-based approach and its implementation as the tool called JSeft (JavaScript Event and Function Testing), were introduced by Mirshokraie et al. in 2015 [266]. JSeft generates Document Object Model (DOM) event based test cases as well as test cases for individual JavaScript functions, augmenting the test inputs with automatically generated mutation-based oracles. JSeft was evaluated on 13 JavaScript applications, ranging in size from 206 to 26,908 lines of code [266], comparing JSeft to Artemis, running both for 10 minutes, and recording coverage achieved. On average, JSeft achieved 68.4% statement coverage compared to 44.8% for Artemis (during the 10 minute execution period on the 13 subjects investigated).

Although both Artemis and JSeft potentially advance the state of practice, due to their more intelligent test data generation techniques, neither addresses the problem of UI test sequence length. A fault-revealing test sequence will tend to be more valuable if it is shorter because a shorter test sequence will be easier and quicker for developers to investigate. Shorter sequences would also tend to reduce debugging time compared to longer sequences for the same fault, all else being equal. This motivated authors of testing approaches for other domains to incorporate techniques that specifically aim to reduce the length of fault-revealing test sequences [114, 248, 250, 268]. However, for JavaScript, there is no existing testing system that seeks to reduce test sequence length, while maximising coverage and fault revelation. Furthermore, existing techniques usually use a very limited set of UI events (such as click and type events but not scroll, touch and other arbitrary gestures). Finally, the existing state-of-the-art

[4]http://github.com/cs-au-dk/Artemis

implementations such as Artemis provide a harness for testing JavaScript, but does not produce a separate test suite that can be collected and used to reproduce the tests (and consequently their faults). While the latter constraints are largely engineering implementation details (albeit important ones), the first issue (test sequence length) is a fundamental scientific limitation.

We introduce the OCTOPUZ system as a conceptual replication[5] [343] of our previous work SAPIENZ [248], to address the problem of effective and efficient fully automated JavaScript test data generation, targeting high coverage and fault revelation, while minimising test sequence length.

The primary contributions of this chapter are:

1. The OCTOPUZ test generation system, which implements a comprehensive list of web UI events for effective and efficient practical JavaScript testing. It evolves efficient tests that optimise the three competing objectives of maximising coverage, maximising fault revelation, and minimising test sequence length. Our implementation of OCTOPUZ will be made publicly available upon publication. We will also provide the evaluation dataset to the public, which contains experiment data on 10 real-world non-trivial JavaScript applications.

2. The empirical replication. While replications are an essential and integral component of empirical software engineering [184, 196, 343], the number of replication studies in the software engineering community remains limited. Our experiment results echo the findings of the SAPIENZ work [248]: OCTOPUZ revealed 8 new real faults thereby demonstrating the usefulness of OCTOPUZ (previous related studies seldom reported real faults [38, 250, 266]). We also find OCTOPUZ outperforms the state of the art on coverage and fault revelation, even though it also has to meet its additional objective of minimising test sequence length. This replication demonstrates that the SAPIENZ approach extends from the mobile Android paradigm to the web-based JavaScript paradigm. It offers a similarly strong improvement over the state-of-the-practice/art techniques, in terms of coverage, fault revelation, and minimised test sequence length.

[5]Here 'conceptual replication' is in contrast to 'exact replication'. That is, instead of replicating the same experiments, we test the effectiveness of the approach on another domain (JavaScript testing) in a non-exact way.

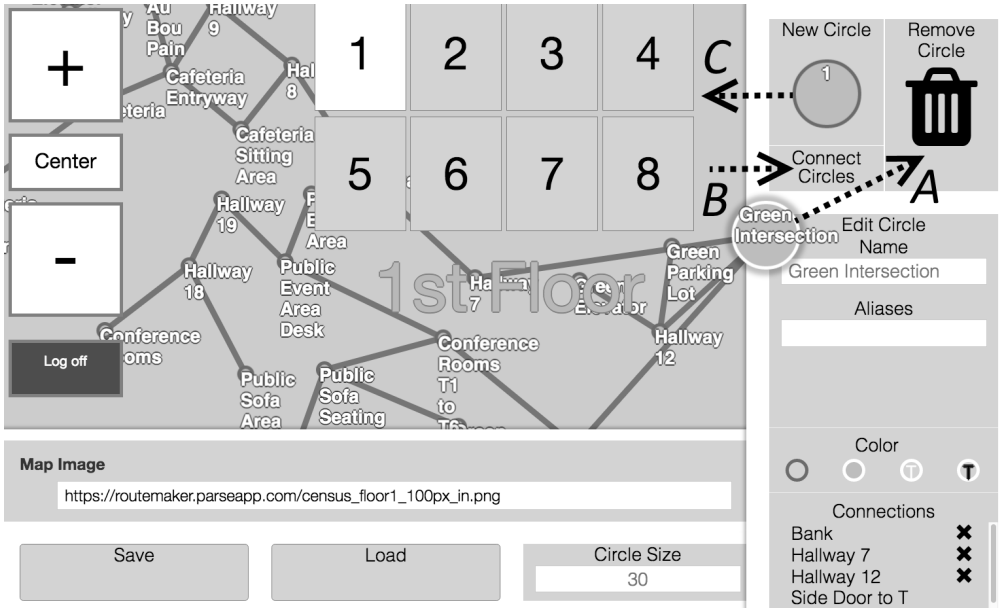



Figure 4.1: A screenshot of JavaScript 'Route-maker'. Interactions: *A. drag a 'Circle' to the 'bin'*; *B. click the 'Connect Circles' button*; *C. drag a 'New Circle' to the map*

```
2352: $.fn.draggable = function() {
  | ...
2359: var start = function(e) {
  | ...
2437: firstCircle = $('.selected-edit');
2441: var firstPos = firstCircle.position();
  | ...
2247: if (noDuplicate == true) {
2248: createLine(firstPos.left, ...);
  | ...
```

Figure 4.2: JavaScript code of Route-maker index.js

## 4.2 Motivation and Challenge

**The motivating example.** Figure 4.1 presents a screenshot of a real-world JavaScript web application named Route-maker. Route-maker is a 2D indoor map editor that is available both on CodePen[6] and GitHub[7]. In order to test the features of the editor, a series of user events are required to cover the JavaScript code. The interaction note in Figure 4.1 shows one such event sequences for testing the 'Connect Circles' feature, which leads to a runtime error of the subject: Initially, when the application interface is loaded, a 'Login' (no login information is required) button needs to be clicked in order to trigger the right editor panel. Subsequently, *Action A* drags a 'circle' to the 'bin' icon; *Action B* clicks the 'Connect Circles' button; *Ac-*

[6] http://codepen.io/OurDailyBread/pen/pjGbVX
[7] http://github.com/PureLeaf/route-maker

*tion C* drags a 'new circle' to the map. This event sequence leads to an uncaught exception, thrown by the browser '`TypeError: Cannot read property 'left' of undefined, index.js:2448`'. According to the stack trace of the exception, we can declare that the error happens when line 2448 (see Figure 4.2) tries to create a line between two 'circles', and `left` is a member of circle position `firstPos`. These statements are part of the `$.fn.draggable/start` function. By checking the JavaScript code shown in Figure 4.2, we can further infer that the error is caused by an real bug due to a situation that is not considered in the code: `firstPos` can be `undefined` if the `firstCircle` target has already been removed in the previous step (*Action A*).

Although this is an apparently obvious fault, it remains a non-trivial task for automated testing techniques to cover: existing JavaScript testing tools Gremlins [251], Artemis [38] and JSeft [266] all failed to cover the faulty line of code, and even the entry to the outer function `$.fn.draggable/start`.

In theory, if allowed a sufficiently long sequence, then random testing can always generate a sequence that will reveal any given fault, but this sequence may, of course, prove to be infeasibly long. In practice, there is normally a budget for testing, which inhibits arbitrarily long test sequences, but the value of a fault-revealing test sequence is loosely proportional to its length; developers may be reluctant to debug a program with respect to a very long fault-revealing test sequence.

**The challenge.** We summarise the challenge of automated testing of interaction-oriented real-world JavaScript application as follows: 1. *Complex interactions:* The JavaScript application may require users to click a target element, press a certain key, scroll down the page, drag an item, or perform gestures in combing a series of complex event sequences. A comprehensive list of event types and combinations of these events may be required to reveal a fault. 2. *Conflicting objectives:* The goal of covering complex interactions is inherently in conflict with the goal of reducing test sequence length. Seeking the minimum test sequence that reveals faults through complex interactions is a non-trivial task. 3. *Fault revelation:* Even for parts of the system that enjoy freedom from faults, it is ideal to perform adequate testing, both to increase confidence in this fault freedom, and to ensure against future fault insertion.

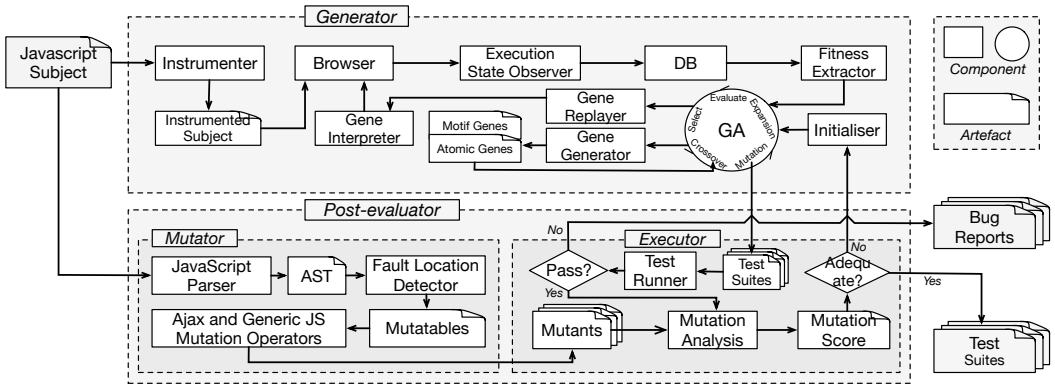



Figure 4.3: The workflow of Octopuz

Therefore, testing purely in terms of fault revelation may be insufficient.

## 4.3 The Octopuz System

We designed the Octopuz system by conceptually replicating the Sapienz approach to generate complex UI interaction data for testing JavaScript applications, while simultaneously optimising multiple testing objectives. During test data generation, fitness of the generated tests is assessed with respect to previously detected (and therefore) known faults, coverage and test sequence length. Similar to Sapienz, Octopuz accepts JavaScript subject as input to the system and generates tests suites and bug reports as the output.

As a conceptual replication of Sapienz, Octopuz has an extra 'post-evaluator' component based on mutation analysis. This enables further assessment on the effectiveness of the generated tests, through a comprehensive set of artificially-seeded faults. These artificial faults are seeded via JavaScript mutation testing techniques, which are especially helpful to inspect the fault detection capabilities of the generated tests on bug-free subjects. To keep consistent with Sapienz, Octopuz does not treat the mutation score as an extra objective. Also, although it may be a useful heuristic to guide the search, incorporating such mutation analysis information into each generation of the search process is computation-intensive [179], thus too time-consuming to be useful in practice. In the later experimental section, we only use the 'post-evaluator' for evaluation purpose and do not further evolve the tests based on the obtained mutation

score.

We assume the subject JavaScript application comes with a UI, usually in the form of HTML pages executed in a browser. When a browser parses a HTML page, it uses a DOM to represent the content, and executes the top level JavaScript code. JavaScript dynamically manipulates the HTML DOM object, thereby enabling interactive features of the subject so we search for user interactions that trigger event handlers. OCTOPUZ automatically generates system level UI test suites that simulate such user interactions. We use the term 'UI test suite' to refer to a set of UI test cases, each of which is a script corresponding to a sequence of UI events.

### 4.3.1 Approach Overview

The overall workflow of OCTOPUZ is presented in Figure 4.3. OCTOPUZ takes a JavaScript subject as input and outputs a set of test-optimised test suites together with a set of error reports generated during the test generation process. The approach consists of three major components: *Generator*, *Mutator* and *Executor*. *Generator* is based on multi-objective optimisation, which aims to generate test suites that maximise coverage and fault revelation, while minimising the length of the test sequences in the suites.

OCTOPUZ uses mutation analysis to enable the assessment of test suite effectiveness, in terms of its ability to kill generated mutants. More specifically (described in Section 4.3.4), *Mutator* seeds a comprehensive set of artificial faults [265, 292] into the JavaScript application and generated mutated versions of the subject, while *Executor* takes the test suites generated by *Generator* and applies them to the mutated subject versions. The mutation score is computed in order to evaluate the effectiveness of the automatically generated test suites. If the generated tests are inadequate according to a predefined threshold of mutation score, they are sent back to the *Generator* for further evolution. These three components work together to generate effective test suites with multiple objectives optimised.

Algorithm 4.1 shows the top-level steps of OCTOPUZ. The iteration finishes when

**Algorithm 4.1:** The Top-level Algorithm of Octopuz

**Input**: Subject $\alpha$, generator function $\gamma$, crossover, mutation, expansion probabilities $\langle \rho, \sigma, \tau \rangle$, max generation $\lambda$, mutator function $\mu$, evaluator function $\epsilon$, mutation score threshold $\theta$, resource $\beta$.
**Output**: Test suites $\mathcal{S}$, error reports $\mathcal{R}$.

1 **function** *octopuz*$(\alpha, \gamma, \mu, \epsilon, \theta, \beta)$
2 $\quad \mathcal{S} \leftarrow \emptyset;\ \mathcal{R} \leftarrow \emptyset;\ \theta' \leftarrow 0;$ ▷ `initialise`
3 $\quad$ // `until exhausted or mutation-adequate`
4 $\quad$ **while** $\beta > 0 \ \wedge \ \theta' < \theta$ **do**
5 $\quad\quad$ // `multi-objective test generator`
6 $\quad\quad \langle \mathcal{S}, \mathcal{R}', \beta \rangle \leftarrow \gamma(\alpha, \mathcal{S}, \beta, \langle \rho, \sigma, \tau \rangle, \lambda);$
7 $\quad\quad \mathcal{R} \leftarrow \mathcal{R} \ \cup \ \mathcal{R}';$
8 $\quad\quad$ // `generate mutants for mutation testing`
9 $\quad\quad \mathcal{A} \leftarrow \mu(\alpha, \beta);$
10 $\quad\quad$ // `evaluate test effectiveness`
11 $\quad\quad \theta' \leftarrow \epsilon(\mathcal{A}, \mathcal{S}, \beta)$
12 $\quad$ return $\langle \mathcal{S}, \mathcal{R} \rangle;$

the assigned resource (e.g, execution time) is exhausted or the generated test suites are mutation-adequate. In the following subsections, we introduce the major steps of Algorithm 4.1 in more detail.

### 4.3.2 Multi-objective JavaScript Testing

Octopuz' *Generator* is based on multi-objective test suite optimisation. The test suite generator follows the process described in Algorithm 4.2. At each generation (lines 5 - 22), the population of test suites is manipulated by a crossover operator (lines 5 - 7) and mutation operator (lines 8 - 10), which increases population diversity. A specially designed 'expansion operator' (line 11) is used to introducing new types of events into the population. Fitness evaluation (line 12) is conducted by executing the test suites and measuring the objectives (via instrumentation). The NSGA-II [95] selection operator (lines 14 - 20) is used to maintain Pareto-optimality of the population by selecting non-dominated individuals to form the next generation. Finally the evolved test suites, revealed errors and the resources remaining are returned (line 23).

**Objectives to optimise.** There are many competing objectives that developers may care about simultaneously when testing JavaScript applications. For example, code coverage, test sequence length, number of found crashes, realism and replicatablity of the test, execution time, and many more [144]. Octopuz is designed to optimise three of these objectives simultaneously: code coverage, fault revelation and test sequence length.

**Algorithm 4.2:** Multi-objective test generator

**Input**: Subject $\alpha$, original test suites $\mathcal{S}$, crossover, mutation, expansion probabilities $\langle \rho, \sigma, \tau \rangle$, max generation $\lambda$, resource $\beta$
**Output**: Evolved test suites $\mathcal{S}$, error reports $\mathcal{R}$, left resource $\beta$

1 **function** $\gamma(\alpha, \mathcal{S}, \beta, \langle \rho, \sigma, \tau \rangle, \lambda)$
2 $\quad \mathcal{R} \leftarrow \emptyset;\ \lambda' \leftarrow 0;$ ▷ initialise
3 $\quad$ // until max generation or run out of resource
4 $\quad$ **while** $\lambda' < \lambda \wedge \beta > 0$ **do**
5 $\quad\quad$ **for** each $\langle \psi_i, \psi_j \rangle \in selectCrossoverIndidual(\mathcal{S}, \rho)$ **do**
6 $\quad\quad\quad$ **for** each $\psi \in crossover(\psi_i, \psi_j)$ **do**
7 $\quad\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\psi\};$
8 $\quad\quad$ **for** each $\psi_i \in selectMutationIndidual(\mathcal{S}, \sigma)$ **do**
9 $\quad\quad\quad$ **for** each $\psi \in mutate(\psi_i)$ **do**
10 $\quad\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\psi\};$
11 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{expandLandscape(\mathcal{S}, \tau)\};$ ▷ new genes
12 $\quad\quad \langle \mathcal{F}, \mathcal{R}' \rangle \leftarrow evaluateFitness(\mathcal{S});$ ▷ fitness evaluation for $\vec{x} \in \mathcal{S}$ (see Equation 4.1)
13 $\quad\quad \mathcal{R} \leftarrow \mathcal{R}' \cup \mathcal{R}';$ ▷ update error reports
14 $\quad\quad \mathcal{S}' \leftarrow \emptyset;$ ▷ for non-dominated individuals
15 $\quad\quad$ **for** each front $\mathcal{P} \in nonDominateSort(\mathcal{S}, \mathcal{F})$ **do**
16 $\quad\quad\quad$ **if** $|\mathcal{S}'| \geq |\mathcal{S}|$ **then** break;
17 $\quad\quad\quad \mathcal{D} \leftarrow getCrowdingDistance(\mathcal{P});$
18 $\quad\quad\quad \mathcal{S}' \leftarrow \mathcal{S}' \cup \mathcal{P};$
19 $\quad\quad$ // selection based on fitness and crowding distance
20 $\quad\quad \mathcal{S} \leftarrow select(\mathcal{S}', \mathcal{F}, \mathcal{D});$
21 $\quad\quad \beta \leftarrow update(\beta);$ ▷ update resource left
22 $\quad\quad \lambda' \leftarrow \lambda' + 1;$ ▷ next generation
23 $\quad$ return $\langle \mathcal{S}, \mathcal{R}, \beta \rangle;$

**Genetic representation.** Octopuz performs whole test suite evolution [26, 115]. It regards a test suite as a genetic individual. Each genetic individual contains multiple test event sequences. For each event sequence, an event is a genetic gene which can be either a low-level atomic gene or a high-level motif gene. A low-level atomic gene executes an atomic (micro) action, such as `click`, `type`, `scroll`, a full list of which can be found in the W3C's UI Events Specification [376]. By contrast, a high-level motif gene executes a series of (macro) actions, following a customisable motif pattern, such as `fill-form-and-submit` or `drag-item-and-release`[8].

The atomic and motif genes are generated by a *JavaScript event generator* (which is implemented via the W3C's API of `dispatchEvent`[9]). Octopuz represents a gene as a tuple, including event type, event parameters (such as the location of the event or its associated DOM nodes), the associated form string values, and information on the running environment, e.g., the browser states.

**Genetic operators.** Octopuz uses 4 genetic operators to guide the search: crossover,

[8] In our evaluation we use a single generic motif pattern that does not incorporate any human intelligence nor any domain knowledge (to avoid biasing the evaluation).

[9] `http://www.w3.org/TR/uievents/#event-flow`

mutation, expansion, and selection. Crossover and mutation are two variation operators to alter the composition of genetic individuals. OCTOPUZ uses uniform crossover and a customised mutation operator by swapping gene positions. JavaScript applications involve many different kinds of events; failing to generate one of these event types may harm coverage.

Since it is possible that the initial population generated by OCTOPUZ may not cover a comprehensive set of event types, to further explore the landscape of the search, we propose to use an expansion operator that increases the diversity of the genetic population. The expansion operator generates new individuals that bring new genes into the population. Expansion is thus a form of mutation operator, although it does not change individual to which it is applied, so we distinguish it from 'pure mutation', which does. For the selection operator, we use the Pareto-ranking of NSGA-II proposed by Deb et al. [95], in order to maintain Pareto-optimality of the generated test sequences regarding all three objectives.

**Fitness evaluation.** OCTOPUZ represents the fitness of an individual as a tuple, consisting of the values for the three objectives, namely coverage, the number of revealed errors, and length of the test:

$$Fitness(\vec{x}) = \langle Coverage(\vec{x}), \#Error(\vec{x}), Length(\vec{x}) \rangle \tag{4.1}$$

In above equation, the coverage, number of errors and length are the accumulated value for each test cases inside the test suite $\vec{x}$. In order to increase scalability, we exploit the inherent parallelism of SBSE [39, 270, 401], performing multiple fitness evaluations concurrently.

### 4.3.3 JavaScript Test Oracles

Despite the recent progress [120, 127, 173], the oracle problem remains largely unsolved [49]. Automated test oracles can be classified into three types [49]: specified, derived and implicit oracles. OCTOPUZ uses implicit oracles, based on the implicit knowledge that 'JavaScript runtime errors (such as uncaught exceptions and null pointer refer-

Table 4.1: Mutation operators in AjaxMutator

| Feature | Defect class | Component |
|---|---|---|
| Event-driven model | User event | Other target element |
| | | Other event type |
| | | Other callback |
| | Time event | Other duration |
| | | Other callback |
| Asynchronous communication | Request | Other URL |
| | | `GET` if `POST`, and vice versa |
| | Response | Failure callback |
| | | Other success callback |
| | | Empty message body |
| DOM manipulation | Selection | Parent or child element |
| | Structure | Src. and dst. elements at append and replace statements |
| | Element | Creation statement |
| | | Removal statement |
| | | Cloning statement |
| | | Normalization statement |
| | Attribute | Other attribute assigned |
| | | Other value assigned |

ences) are nearly always errors and need to be fixed'. JavaScript console errors are also obtained via the console of most modern browsers. These console error messages (including error type and stack trace) have been widely used for studying real-world JavaScript faults [295, 296, 297, 406]. Octopuz also leverages such console errors as test oracles. When counting the number of revealed errors, we calculate the number of 'unique errors' by checking the error type and stack trace provided by the console error message.

### 4.3.4 JavaScript Mutation Analysis

Mutation analysis [178] inserts seeded faults into a software system, in the hope that test cases that reveal them may also reveal real faults, which is supported by empirical evidence [31, 185]; this technique injects artificial faults into a program under test and checks whether a test suite detects these injected faults. Octopuz uses a publicly available[10], mature technique for JavaScript mutation analysis [298], to mutate JavaScript programs using JavaScript-specific mutation operators. Table 4.1 shows the

[10] `http://github.com/knishiura-lab/AjaxMutator`

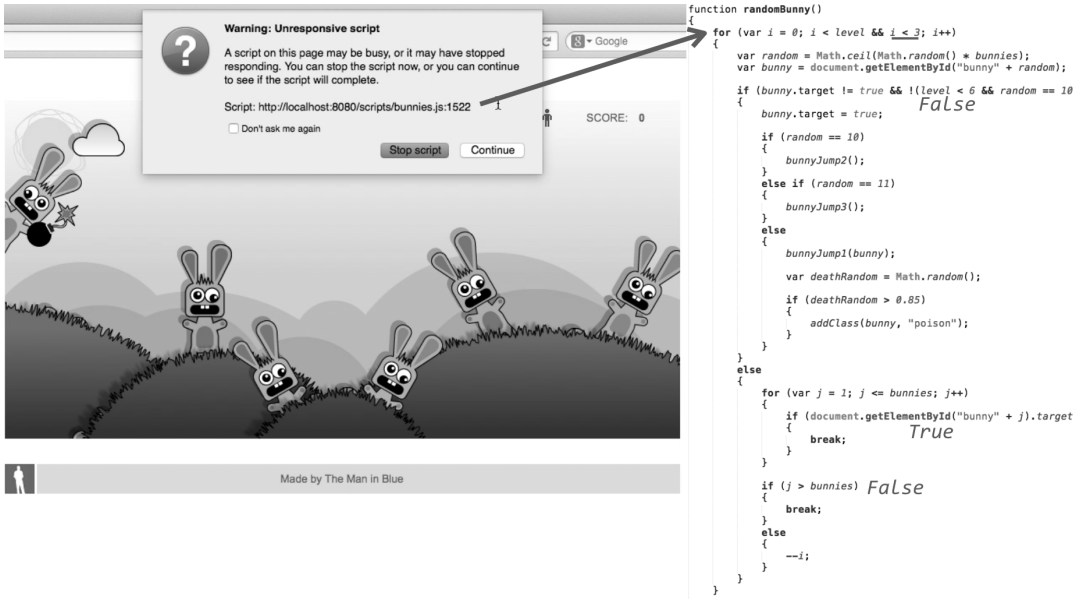



Figure 4.4: Automated testing of a real-world JavaScript gaming app named Bunny-Hunt, and one unresponsive defect revealed by Octopuz (where the defect is an infinite loop caused by a `i++/i--` pair as shown in the code snippet)

operators that cover Ajax mandatory features, such as an event-driven model, asynchronous communication, and dynamic DOM manipulation [156]. AjaxMutator also implements the general JavaScript mutation operators proposed by Mirshokraie et al. [265]. Octopuz involves AjaxMutator for runtime test optimization, in a way that evaluates the effectiveness of the test suites and eliminate those weak ones through potential multiple rounds of evolution (see Figure 4.3).

## 4.4 Implementation

We implemented the approach used by Sapienz [248] for the Octopuz test generation system. Given a subject application as input, Octopuz instruments its source code using JSCover[11]. It tests on the instrumented subject according to the Octopuz algorithm defined in the previous section, to yield a set of optimised test suites together with reports showing the detected runtime errors and their stack traces. Each test case is a sequence of UI events, each of which is scripted with its event type and associated parameters, such as its location in the browser's window. The objective properties of each individual test suite constructed during the entire evolutionary search process

[11] http://tntim96.github.io/JSCover

are also provided in a logbook. The logbook is used to form a Pareto front of non-dominated test suites, from which developers can choose, according to their priorities on each objectives.

The generator component of OCTOPUZ implements NSGA-II for multi-objective search, based on the DEAP Python framework [112]. OCTOPUZ automates browsers using Selenium and it evaluates fitnesses in parallel for each individual in the population. The browsers (across different platforms) supported by Selenium can also be used by OCTOPUZ.

For the bottom layer of web UI event generation, OCTOPUZ extends gremlins.js to enable motif event, instrumentation and test script generation and replay. Five types of web UI events are supported, including `click`, `scroll`, `press key`, `fill form` and `motif`. A specifically-designed motif pattern might elevate the coverage, however this would lead to an unfair evaluation (because it would give OCTOPUZ specific domain knowledge not available to other techniques against which we compare). Therefore, for evaluation purposes, we implemented a simple universal pattern that applies to any JavaScript applications; it fills text fields and then clicks clickable elements, to simulate the generic `fill-form-and-submit` usage pattern. For the mutator component, we used the AjaxMutator tool that implements JavaScript-specific mutation operators (Section 4.3.4).

## 4.5 Empirical Evaluation

To evaluate our OCTOPUZ system, we measure its ability to generate test suites that maximise code coverage and fault revelation, while minimising test sequence length. There is, to our best knowledge, no state-of-the-art technique that seeks to achieve high coverage, high fault revelation and low test sequence length simultaneously for JavaScript testing, as OCTOPUZ does. Nevertheless, there are recently published research techniques for generating tests that seek to find faults and achieve high coverage.

These are the tools JSeft [266] and Artemis [38]. Both tool implementations that are publicly available do not output replay test scripts that can be measured for test

sequence length. JSeft and Artemis attempt to intelligently cover the JavaScript code, thereby achieving high coverage and fault revelation, and either could therefore be thought of as the current state of the art with respect to these two objectives. We can measure their coverage and the number of faults they detect, using our automated oracle, and so we compare OCTOPUZ against these two techniques for the two objectives of fault revelation and code coverage.

This is a slightly unfair comparison; it punishes our approach, OCTOPUZ, because OCTOPUZ seeks to reduce test sequence length, which is inherently in conflict with the objective of achieving higher coverage and higher fault revelation. A longer sequence, all else being equal, has a greater probability of achieving greater coverage and fault revelation than a shorter one, so our evaluation favours the existing state of the art, JSeft and Artemis, when we focus solely on coverage and fault revelation without taking into account test sequence length. Nevertheless, a developer may be primarily interested in finding faults and achieving coverage, and therefore it seems reasonable to compare OCTOPUZ with both JSeft and Artemis. If OCTOPUZ is notably worse for either coverage or fault revelation, then a developer might reasonably reject it, despite its ability to produce reduced test sequence length.

There is also a widely-used tool called 'Gremlins' [251], which is available on GitHub and has over 6,660 stars/likes, so can be regarded as a practical tool against which to compare as a baseline. When comparing against Gremlins, we can obtain its generated test events, and thereby measured the length of test cases as well as the coverage and fault revelation of the tests generated. This affords a direct head-to-head comparison with OCTOPUZ. However, this comparison can only be regarded as a baseline, since Gremlins is essentially a random testing tool, and one would expect that an intelligent search ought to be capable of outperforming a random search.

Therefore, replicating the structure of our evaluation of SAPIENZ on Android, our evaluation of the OCTOPUZ system consists of two overall research questions, the first of which compares OCTOPUZ with the 'state-of-practice' tool Gremlins (according to coverage, fault revelation and test sequence length), while the second addresses the comparison with the state-of-the-art research techniques (comparing the performance

of OCTOPUZ for coverage and fault revelation with JSeft and Artemis).

**RQ1 (Comparison to the state of practice):** How does the performance of our technique, OCTOPUZ, compare with that of the popular state-of-practice tool Gremlins?

We compare the tests generated by OCTOPUZ and Gremlins according to three objectives, giving three sub-RQs:

**RQ1.1 (Coverage):** How does the coverage achieved by OCTOPUZ compare with the coverage achieved by Gremlins?

**RQ1.2 (Fault revelation):** How does the fault revelation achieved by OCTOPUZ compare with that achieved by Gremlins?

**RQ1.3 (Test sequence length):** How much smaller are the test sequences produced by OCTOPUZ compared to those produced by Gremlins?

We answer all three of these research questions by applying both OCTOPUZ and Gremlins to the 10 subject applications described in Section 4.5.1. We assess fault revelation using the automated oracle described in Section 4.3.3. We execute OCTOPUZ and Gremlins 30 times each in order to collect a distribution of results, allowing us to compare the performance of the two approaches using inferential statistical techniques. To ensure a fair comparison, we allow Gremlins exactly the same amount of execution time and number of runs of the tool as OCTOPUZ.

Our second set of research questions compares the performance of OCTOPUZ with the two state-of-the-art JavaScript testing techniques JSeft and Artemis:

**RQ2 (Comparison to the state of the art):** How does the performance of OCTOPUZ compare with the performance of the state-of-the-art techniques JSeft and Artemis?

For these two techniques and OCTOPUZ, we collect information concerning fault revelation and coverage achieved, for a 10 minute execution of each tool on each of the 10 subject applications described in Section 4.5.1. Since the tools of JSeft and Artemis do not permit us to control the number of test cases generated, nor their test sequence length, the fairest way to compare with the two technique seems to allow both the same

amount of wall clock time. This would best reflect the perspective of a developer who would be interested to find faults and to achieve coverage as quickly as possible. We allow 10 minutes execution time per subject, which echoes the study of JSeft [266].

**RQ2.1 (Coverage):** How does the coverage achieved by OCTOPUZ compare with the coverage achieved by each of JSeft and Artemis in a 10-minutes execution of each technique on each of the 10 subject applications?

**RQ2.2 (Number of faults revealed):** How many faults are revealed by OCTOPUZ compared with the faults revealed by each of JSeft and Artemis in a 10-minutes execution of each technique on each of the 10 subject applications?

For fault revelation, we measured the faults detected by the automated oracle (as with RQ1). However, not all of these real-world JavaScript applications contain faults that are revealed in this way. This does not mean that they are fault-free, but merely that the automated oracle is insensitive to the faults they may contain. We therefore augment an analysis of real faults, as revealed by the automated oracle, with a mutation testing analysis [178], by seeding artificial faults into those applications for which the automated oracle reveals no faults. We use the JavaScript mutation tool AjaxMutator [292] to seed faults. We first use its mutators to generate mutants (each mutant contains a single fault) of the subject. Then we randomly select 20 mutants for each subject.

**RQ2.3 (Complementarity):** What is the degree of overlap between the faults revealed by OCTOPUZ and those revealed by each of JSeft and Artemis?

We are interested in complementarity, because even should we find that OCTOPUZ outperforms the state of the art, there may, nevertheless, remain some faults found by one technique that are not found by the other. In such a situation, the JavaScript developer might choose to execute all test tools to find the maximum number of faults overall. RQ2.3 therefore investigates the degree to which each of the techniques is complementary to the other, in terms of the overlap between faults found; the lower the overlap, the greater the complementarity.

Table 4.2: JavaScript application subjects

| Subject | SLOC | Source | URL |
|---|---|---|---|
| Route-maker | 2,836 | CodePen | http://codepen.io/OurDailyBread/pen/pjGbVX |
| Javascript-Clipper | 4,822 | CodePen | http://codepen.io/timo22345/pen/GpgjNQ |
| Pong | 5,149 | CodePen | http://codepen.io/Glucio/pen/wazMMX |
| Self-code | 6,821 | CodePen | http://codepen.io/myersg86/pen/JdwaxN |
| Spotify-vue | 7,019 | CodePen | http://codepen.io/tjFogarty/pen/RrvMBO |
| Guess-the-number | 10,575 | CodePen | http://codepen.io/sevenstreak/pen/pJBoOw |
| BunnyHunt | 695 | JSeft | http://themaninblue.com/experiment/BunnyHunt |
| Homeostasis | 3,203 | Artemis | http://brics.dk/artemis/examples |
| Javascript-tetris | 314 | GitHub | http://github.com/jakesgordon/javascript-tetris |
| 2048 | 662 | GitHub | http://github.com/gabrielecirulli/2048 |

### 4.5.1 Subject Applications

We use 10 real-world JavaScript applications as the subjects for evaluation. This dataset consists of two subjects from the general open-source community GitHub ('2048', one of the most popular JavaScript application with over 7,998 stars and 13,953 forks; and 'Javascript-teris', the less popular HTML5 JavaScript application), six subjects from the front-end-specific open-source community CodePen and two randomly selected subjects from the Artemis and JSeft studies for a 'sanity check' (to replicate and recalibrate against the subjects previously used). The six subjects from CodePen are selected by following the systematic steps below: We first collect all front-end applications by using queries: 'JavaScript OR Ajax', which leads to 90,296 applications in total. Since we aim to evaluate on non-trivial subjects, we filter out those applications with fewer than 1,000 lines of code. We sort the remaining applications by the number of source lines of code, and include the largest applications that meet the following three criteria 1) They exhibit an interaction-oriented design; 2) They use JavaScript on the client-side; 3) They can be executed in our experimental environment.

More detailed information on the 10 subjects can be found in Table 5.1. The 'SLOC' column reports the number of source lines of JavaScript code (excluding comments and libraries such as jQuery), which ranges from 314 to 10,575 lines of JavaScript code. The source of the subjects and their publicly accessible URLs are also provided in the

Table 4.3: Vargha-Delaney $\hat{A}_{12}$ effect size on the comparison of Octopuz to Gremlins

| Subject | Coverage | | #Errors | | Length | |
|---|---|---|---|---|---|---|
| | **p** | $\hat{A}_{12}$ | **p** | $\hat{A}_{12}$ | **p** | $\hat{A}_{12}$ |
| 2048 | <0.01 | 1.00 | >0.99 | – | <0.01 | 1.00 |
| BunnyHunt | <0.01 | 0.73 | >0.99 | – | <0.01 | 1.00 |
| Guess-the-number | <0.01 | 1.00 | >0.99 | – | <0.01 | 1.00 |
| Homeostasis | 0.55 | – | <0.01 | 0.11 | <0.01 | 1.00 |
| Javascript-tetris | 0.67 | – | >0.99 | – | <0.01 | 1.00 |
| JavascriptClipper | <0.01 | 0.96 | >0.99 | – | <0.01 | 1.00 |
| Pong | <0.01 | 0.91 | >0.99 | – | <0.01 | 1.00 |
| Route-maker | <0.01 | 1.00 | <0.01 | 1.00 | <0.01 | 1.00 |
| Self-code | <0.01 | 1.00 | >0.99 | | <0.01 | 1.00 |
| Spotify-vue | <0.01 | 1.00 | <0.01 | 1.00 | <0.01 | 1.00 |

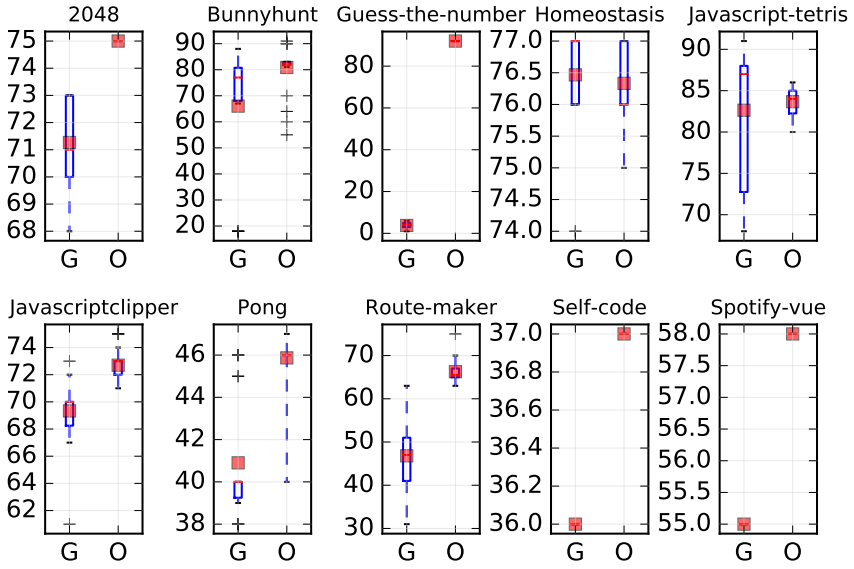
– indicates a statistically non-significant result (at the 0.05 level).

table.

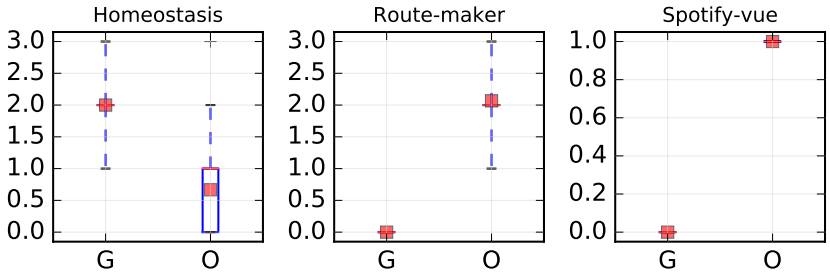
### 4.5.2 Experimental Setup

We conduct all experiments on a machine with an Intel Xeon 3.5GHz CPU and 16GB memory, running Ubuntu 14.04. For each technique under evaluation, we use its latest publicly available version, at the time of writing. We assign 10 minutes to each tool for each subject. All experiments for RQ1 were repeated 30 times to provide a sample of runs for statistical analysis. We use the non-parametric inferential statistical significance test, the Wilcoxon signed rank test [387], for comparing Octopuz with the state-of-practice tool Gremlins. The test is applied at the 0.05 alpha level, and the Vargha-Delaney [369] $\hat{A}_{12}$ statistics is used to measure an untransformed [290] Vargha-Delaney effect size, because we are simply interested in the likelihood that one approach will outperform another on an arbitrary, unknown scenario. The differences between approaches are characterised as 'small', 'medium' and 'large' when the $\hat{A}_{12}$ effect size exceeds 0.56, 0.64, and 0.71, respectively [35, 149].

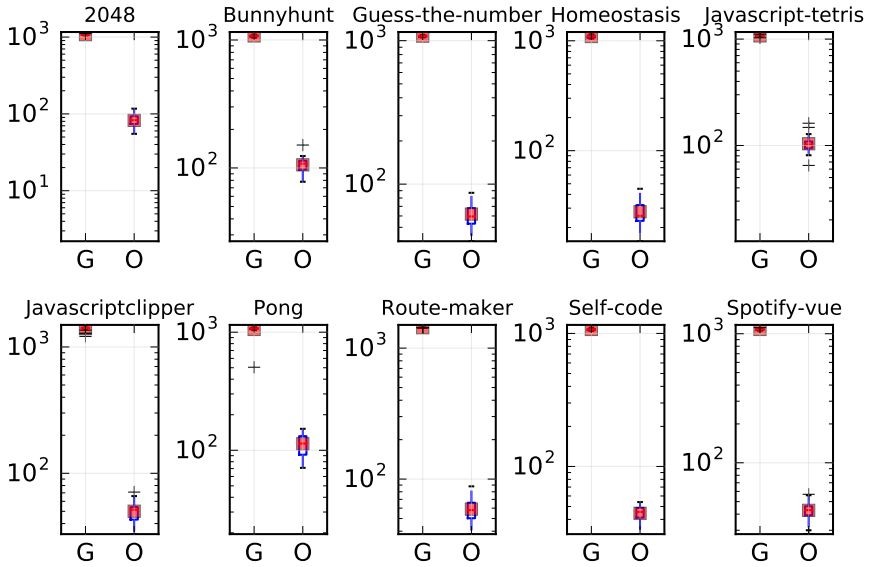
We use the browser-reported console error as the automated test oracle and use JSCover to measure code coverage for each tool. Note that, since Artemis is tightly coupled with its integrated WebKit browser, we have no control over its browser's behaviour and thus cannot measure its coverage with JSCover directly. For a fair comparison, when




(a) Boxplots of statement coverage in percentage. ('G' for Gremlins and 'O' for Octopuz. The square symbol indicates the mean value.)




(b) Boxplots of number of revealed runtime errors. (Boxplots for the subjects where no runtime errors were revealed are ignored.)




(c) Boxplots of length of generated test sequences.

Figure 4.5: Performance comparison of Octopuz versus Gremlins on 10 subjects

reporting the performance on code coverage, we therefore logged its reported covered lines and convert the line coverage into statement coverage according to the 'statement' criteria used by JSCover.

We report details of our configuration choices for the state-of-the-art and state-of-practice tools in order to support replication, and better understanding of the results: Artemis supports multiple execution modes. In our experiments we used the major mode of 'artemis' [38] and set its strategy priority to 'coverage', because we want to give it the best chance to perform optimally for the coverage assessment criterion. We run Artemis with its own tightly coupled WebKit browser (since Artemis cannot be run with an external browser), and run Gremlins, JSeft and OCTOPUZ with the Firefox 31.8.0 browser. For Gremlins, we use its default uniform distribution when generating different types of events. For JSeft, since all techniques under evaluation should be fully automated, for a fair comparison, we run its dynamic exploration phase automatically (without human intelligence for defining clickable HTML tags for a specific subject). Instead, we use a list of comprehensive clickable tags that are used in its implementation. For OCTOPUZ, we set its crossover, mutation, and expansion rate to 0.4, 0.3, and 0.2, respectively. We limit the maximum number of generations to 100, with population size of 10, and with each individual containing 3 test sequences. These parameters remain the same through all experiments and are not tuned specifically for any certain subjects.

### 4.5.3 Experimental Results

#### Comparison to the State of Practice

**RQ1 (Overall statistics).** Table 4.3 lists the Vargha-Delaney $\hat{A}_{12}$ effect size for the three objectives, coverage, the number of runtime errors and a mean of the length of generated test scripts. For each objective, the columns contain the effect size measure comparisons for OCTOPUZ and Gremlins. As listed in the table, OCTOPUZ significantly outperforms Gremlins with large effect size on 8 out of 10 subjects for coverage and on 10 out of 10 subjects for event sequence length (with large effect size). OCTOPUZ

found errors in 3 out of the 10 subjects, while Gremlins found errors in only 1 subject. Next, we present more detailed results regarding each sub research question of RQ1.

**RQ1.1 (Coverage).** The statement coverage results from running Gremlins and Octopuz on 10 subjects for 30 runs are illustrated as a bar chart in Figure 4.5 (a). The boxplots in the chart illustrate the observed coverage on each subject, across a total number of 30 runs.

Octopuz outperforms Gremlins on 9 out of 10 subjects in terms of mean coverage, though the differences on the coverage are not large. We were surprised by the high coverage achieved by Gremlins, noticing that it outperformed the existing state-of-the-art techniques, with which it has not previously been compared in the literature. This could reflect the greater engineering effort directed towards this industrial strength tool. Indeed, similar observations were made for Android testing [84], for which the industry-strength random testing tool 'Android Monkey' outperformed the state-of-art in 2015 (though it has since been surpassed in 2016 [248]).

On the 'Guess-the-number' subject, Octopuz achieves higher coverage than Gremlins. An observation we attribute to is the fact that random search is inherently ill-suited to 'guessing magical values' such as required for guessing the number, whereas more intelligent search can use fitness guidance to cover such problems. Also, the motif event used by Octopuz enables the guessing behaviour in an efficient way. There are also subjects such as Pong and Self-code, for which neither Octopuz nor Gremlins achieves high coverage. We manually checked these cases and observed that these subjects include extra API functions that are copied from third party libraries. These extra functions are dead code that artificially prevent a high coverage. Note that, although we excluded external libraries when measuring statement coverage, we did not specifically excluded those copy-pasted into the subject.

**RQ1.2 (Fault Revelation).** The number of revealed runtime errors for each run is depicted in Figure 4.5 (b). Across all 10 subjects, Octopuz in general found more runtime errors for each run when compared to Gremlins. Octopuz revealed faults in 3 of the subjects (Homestasis, Route-maker and Spotify-vue), while Gremlins uncovered only Homestasis' faults. Furthermore, although Gremlins found more 'faults' than

OCTOPUZ on this subject in each run, the two runtime errors it revealed are actually caused by the same underlying fault (Error ID1 and ID2 in Table 4.4).

Collectively, over all 30 runs, we list all detected distinct[12] runtime errors in Table 4.4. As listed in the table, Gremlins revealed 2 runtime errors, where error ID1 is also covered by OCTOPUZ. Note that multiple runtime errors may be caused by a single fault (e.g., errors (ID1,ID2), detected by Gremlins, which are caused by the same fault and error (ID3,ID4), detected by OCTOPUZ, which are also caused by a single fault).

The two testing tools collectively revealed 11 runtime errors that corresponds to 8 unique faults. OCTOPUZ revealed all of these 8 faults, while Gremlins revealed 1 of them. For the remaining 7 subjects, no runtime error was revealed by either tool. Overall, the number of faults detected in our study is larger than those reported in previous studies [38, 266].

**RQ1.3 (Length).** The lengths of Gremlins and OCTOPUZ test scripts are reported in Figure 4.5 (c). The script length generated by Gremlins ranges from 1,039 to 1,431, while for OCTOPUZ, its range is at least one order of magnitude shorter, ranging from 28 to 113. Taken together with the results from RQ1.1 and RQ1.2, we find strong evidence that OCTOPUZ achieves its primary goal of reducing test sequence length, while remaining highly competitive, even surpassing fault revelation and coverage by comparing with the current state of practice. This is our particularly important finding that the current state of practice is highly competitive with the state of the art, hitherto reported in the literature.

### Comparison to the State of the Art

**RQ2.1 (Coverage).** Figure 4.6 reports the performance on statement coverage achieved by Artemis, JSeft, and OCTOPUZ. Across the 10 subjects, OCTOPUZ achieved a mean coverage of 69%, which outperforms that obtained by Artemis (31%) and JSeft (46%). OCTOPUZ' coverage ranges from 37% (Self-code) to 92% (Guess-the-number) and OC-

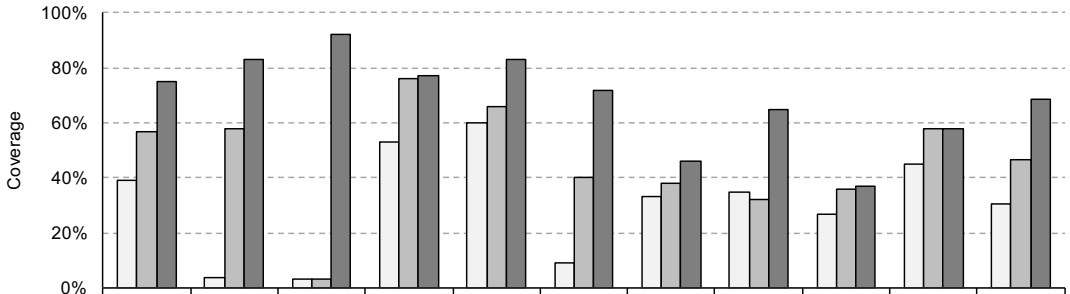
[12]We say a runtime error is distinct if its stack trace together with its error message differs from others.

Table 4.4: All revealed distinct JavaScript runtime errors (30 runs)

| Tool | ID | Subject | JS File | Method | Line | Error Message |
|---|---|---|---|---|---|---|
| Gremlins | 1 | Homeostasis | homeostasis.js | `cheB/that.bound` | 1,060 | `methylNeighbor` is not defined |
| | 2 | | homeostasis.js | `cheB/that.targeting` | 1,065 | `that.methylNeighbor.column` is null |
| Octopuz | 1 | Homeostasis | homeostasis.js | `cheB/that.bound` | 1,060 | `methylNeighbor` is not define |
| | 2 | | homeostasis.js | `molecule/hideDescription` | 157 | `that.keyItem(...)` is undefined |
| | 3 | | homeostasis.js | `molecule/that.mouseIn` | 125 | `moleculeKey.itemhash[that.type]` is undefined |
| | 4 | | homeostasis.js | `molecule/that.mouseOut` | 133 | `moleculeKey.itemhash[that.type]` is undefined |
| | 5 | | flux.js | `flux.canvas/mouseEvent/<` | 1,131 | `mote[("mouse" + event)]` is not a function |
| | 6 | Route-maker | index.js | `returnColor` | 3,657 | `colorData[colorName]` is undefined |
| | 7 | | index.js | `setTextBorderColor` | 3,383 | `textShadow` is undefined |
| | 8 | | index.js | `$.fn.draggable/move` | 2,524 | `circle` is undefined |
| | 9 | Spotify-vue | index.js | `audioTimeUpdated` | 11,515 | `track` is null |

Table 4.5: Comparison on revealed distinct JavaScript runtime errors

| Tool | ID | Subject | JS File | Method | Line | Error Message |
|---|---|---|---|---|---|---|
| Artemis | 1 | 2048 | tile.js | `Tile` | 2 | 'undefined' is not an object (evaluating '`position.x`') |
| | 2 | Javascript-Clipper | index.html | `popup_path` | 1,955 | 'undefined' is not an object (evaluating '`scaled_paths[fr].join`') |
| | 3 | Route-maker | index.js | `setTextBorderColor` | 3,383 | `textShadow` is undefined |
| | 4 | | index.js | `returnColor` | 3,659 | 'undefined' is not an object (evaluating '`colorData[colorName].hex`' |
| JSeft | 1 | Homeostasis | homeostasis.js | `cheB/that.bound` | 1,060 | `methylNeighbor` is not defined |
| | 2 | Route-maker | index.js | `loadData/<.success` | 481 | `mapData[floorName]` is undefined |
| Octopuz | 1 | Homeostasis | homeostasis.js | `cheB/that.bound` | 1,060 | `methylNeighbor` is not defined |
| | 2 | Route-maker | index.js | `returnColor` | 3,657 | `colorData[colorName]` is undefined |
| | 3 | | index.js | `setTextBorderColor` | 3,383 | `textShadow` is undefined |
| | 4 | Spotify-vue | index.js | `audioTimeUpdated` | 11,515 | `track` is null |




| | 2048 | BunnyHunt | Guess-the-number | Homeostasis | Javascript-tetris | JavascriptClipper | Pong | Route-maker | Self-code | Spotify-vue | MEAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Artemis | 39% | 4% | 3% | 53% | 60% | 9% | 33% | 35% | 27% | 45% | 31% |
| Jseft | 57% | 58% | 3% | 76% | 66% | 40% | 38% | 32% | 36% | 58% | 46% |
| Octopus | 75% | 83% | 92% | 77% | 83% | 72% | 46% | 65% | 37% | 58% | 69% |

Figure 4.6: Coverage performance in comparison with Artemis and JSeft

Table 4.6: Fault revelation on real and seeded faults

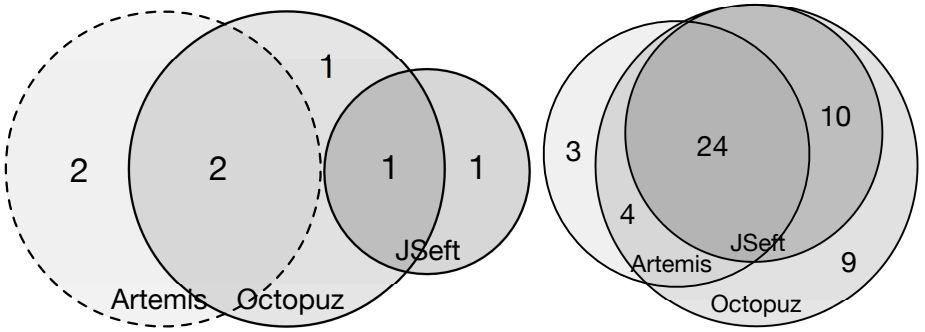
| **Subject** | **Acutal Faults** | | | **Seeded Faults** | | |
|---|---|---|---|---|---|---|
| | **A** | **J** | **O** | **A** | **J** | **O** |
| 2048 | 1 | 0 | 0 | - | - | - |
| BunnyHunt | 0 | 0 | *0 | 4 | 14 | 14 |
| Guess-the-number | 0 | 0 | 0 | 0 | 0 | 7 |
| Homeostasis | 0 | 1 | 1 | - | - | - |
| Javascript-tetris | 0 | 0 | 0 | 13 | 8 | 13 |
| JavascriptClipper | 1 | *0 | 0 | - | - | - |
| Pong | 0 | 0 | 0 | 7 | 6 | 7 |
| Route-maker-v0-3 | 2 | 1 | 2 | - | - | - |
| Self-code | 0 | 0 | 0 | 7 | 6 | 6 |
| Spotify-vue | 0 | 0 | 1 | - | - | - |
| SUM | 4 | 2 | 4 | 31 | 34 | 47 |

'A': Artemis; 'J': JSeft; 'O': OCTOPUZ; '-': not applicable.
'*': an 'unresponsive script' warning was observed.

TOPUZ outperforms Artemis and JSeft for coverage on all subjects except Spotify-vue[13]. We conclude that there is strong evidence that OCTOPUZ outperforms the coverage achieved by current state-of-the-art techniques.

**RQ2.2 (Number of faults revealed).** The overall number of errors revealed is reported in Table 4.6: Both Artemis and OCTOPUZ found 4 runtime errors across the 10 subjects. JSeft found 2 runtime errors. Before reporting on any errors automatically found by a tool (according to our automated oracle), we carefully and manually checked

[13]However, OCTOPUZ uncovered a fault on this subject, not found by either Artemis or JSeft, as we shall see (RQ2.2).




(a) Real faults. (b) Seeded faults.

Figure 4.7: Complementarity on revealed faults

the stack trace information and analysed the JavaScript programs, in order to exclude any false positives. For JSeft and Octopuz, revealed runtime errors are caused by missed null/undefined variables checking. While for Artemis, we can only confirm that 2 of the 4 runtime errors are caused by real faults; the other 2 may be false positives (as will be discussed in RQ2.3). We also observed two cases where 'unresponsive script' warnings were raised, one from JSeft (on 'JavascriptClipper') and one from Octopuz (on 'BunnyHunt', see Figure 4.4).

The last three columns of Table 4.6 presents the result of mutation testing analysis, on the 5 subjects where no runtime errors had been revealed. The total number of 31, 34, and 47 mutants were killed by Artemis, JSeft, and Octopuz, respectively. Note that not all seeded faults would cause runtime errors detectable by our automated oracle. This may explain why the percentage of killed mutants is not very high for any subject/tool.

**RQ2.3 (Complementarity).** Table 4.5 lists the details of the real-fault (not seeded) errors revealed by each technique. Figure 5.13 (a) presents Venn diagram that shows that overlap. As can be seen, Octopuz revealed one error (fault) that neither Artemis nor JSeft found. JSeft revealed one runtime error that was not detected by either of the other tools, while Artemis revealed two such 'runtime errors'.

However we cannot confirm that these two (revealed by Artemis) are caused by real faults. Because we could only manually check the stack traces for JSeft and Octopuz. That is, Artemis provides top-level error messages instead of the full stack trace of an

error. By checking these general error information, we can only infer the likely causes of any errors 'detected'.

We therefore inspected these errors in the 'manual' mode offered by the Artemis tool, where we found the graphical features of the '2048' and 'JavascriptClipper' subjects do not appear to correctly interact with Artemis' instrumented WebKit browser. For instance, in the *2048* case, the initial 'tiles' cannot be shown by Artemis (see Figure 4.8).

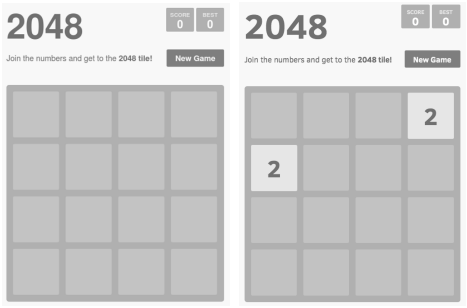
It is not clear whether this should be regarded as a true positive, since the same code executes correctly in the Firefox browser. A similar situation applies to the 'Javascript-Clipper' 'error' reported by Artemis. As such, it is possible that these two 'errors' may be caused by the instrumented WebKit browser that was released in 2011, i.e., it may no longer be compatible with current JavaScript applications (by contrast Firefox 31.8.0 was released in 2015). These observations also highlight the issues that can occur when web testing technologies require a specifically tailored browser as part of the test environment.

The mutation testing results, as illustrated in Figure 5.13 (b), show that Artemis exclusively revealed 3 seeded faults. The 34 mutants killed by JSeft were covered by Octopuz. Octopuz also found 9 faults that neither Artemis nor JSeft managed to cover. Overall, our results on real and seeded faults suggest that the three techniques may complement one the other. The results also indicate that, despite the minimised test sequence length, Octopuz remains highly competitive with the state-of-the-art for fault revelation.

### 4.5.4 Limitations and Threats to Validity

**Internal validity threats.** It is possible that the approach under evaluation may perform better when using tuned parameters. To mitigate this threat, we leave our Octopuz untuned when setting up the experiments and use only a generic motif pattern for Octopuz.

The implementation of the tool Artemis spans four years, so it is possible that its




(a) WebKit (Artemis) (b) Firefox (v31.8.0)

Figure 4.8: The potential false positive on the subject 2048 (where the initial 'tiles' are missing in the Artemis WebKit-based browser)

performance differs from the one used in their original publication [38]. In particular, in our experiments we did not observe the 'undefined' runtime error from the 'Homeostasis' subject, according to its log information[14], which is reported in their study. This may be caused by differential Artemis implementations and/or differences in compilation and execution environments.

**External validity threats.** As with any empirical study on a set of subject programs, care is required in generalising from these results. With the aim of supporting generalisation, we chose subjects from multiple sources, including largest subjects from CodePen. Furthermore, we only used strong sound (but implicit) oracles for the evaluation, i.e., the runtime errors/exceptions threw by the browser. It is possible that more and different JavaScript faults would be revealed by more sophisticated oracles.

## 4.6 Summary

This chapter reported on the Octopuz system, a conceptual replication of the Sapienz approach for multi-objective automated testing for Android apps. Octopuz extends the Sapienz approach from Android to JavaScript, generating short test sequences that achieve high coverage and competitive fault revelation for JavaScript applications. We evaluated Octopuz against both the state-of-practice and the state-of-the-art JavaScript testing approaches, on a collection of 10 real-world JavaScript applications.

[14]However we observed a log `WEBKIT SCRIPT ERROR: "TODO: Get error text!"`, where the error text is not available.

The results show its superior coverage and test sequence length, while retaining competitive fault revelation. The performance of OCTOPUZ echoes the effectiveness of the SAPIENZ approach, demonstrated in our previous work on Android testing [248]. Both approaches outperformed the state-of-practice and the state-of-the-art in their corresponding domains for code coverage. Also, real faults were uncovered on non-trivial real-world applications in both cases. We therefore regard the multi-objective search with novel features of motif events and hybrid exploration as an effective way to generate test inputs for real-world software applications for both Android and JavaScript based platforms.

Our future work will be focused on automatically identifying natural motif events and UI exploration strategies to further improve the performance of our multi-objective automated test generation approach.

# Chapter 5

# Polariz: Harnessing Crowd Intelligence to Support Search-based Mobile Testing

In this chapter, we introduce the Polariz approach, which leverages crowdsourcing to improve search-based mobile test automation approaches. Polariz uses a platform that enables crowdsourced mobile testing from any source of app under test (AUT), via any platform, and by any crowd of workers. It generates replicable test scripts based on manual test traces produced by the crowd workforce, and automatically extracts, from these test traces, motif events that can be used to improve search-based mobile testing approaches such as Sapienz.

Our empirical evaluation of Polariz shows that it was able to assist 434 crowd workers from 24 countries to perform 1,350 task assignments. The crowd was deployed to test 9 popular Google Play apps, each has at least 1 million user installs. The automatically learned motif genes improved Sapienz' performance in 6 out of 9 cases, having no effect (positive or negative) on the remaining 3, in terms of achieved app activity coverage.

## 5.1 Introduction

Real world mobile applications (apps) usually have complex user interfaces and user flows. Generating system tests to cover the functionalities behind these flows is a challenging task. According to a recent study conducted on open sourced Android apps with relatively simple user flows [84], current state-of-art automated mobile testing techniques can only cover up to 50% statements of the subjects code structure.

In this chapter, we introduce an approach to use crowdsourcing to support mobile test automation. By mining the manual test event traces generated by the crowd from the general public, we aim to learn from the crowd intelligence underlying these traces, and further use the extracted event Crowdsourcing-based mobile testing is an under studied area [247]. Even for this initial step into crowdsourced trace collection, currently there is no readily available approach that can be used directly to assist crowdsourced remote mobile app testing. To tackle this problem, we first introduce our Polariz platform as part of our Polariz approach. The platform bridges the gap between the general crowd and automated mobile testing. Subsequently, we propose the Polariz crowd motif pattern extraction algorithm, which learns from crowd intelligence embodied in the crowd's collection of manually constructed tests.

The goal of Polariz' motif extraction component is to capture the 'crowd intelligence' in the representation of 'motif patterns' underlying their interaction event traces for testing mobile apps. We define a 'motif pattern' as a common user interaction pattern that learned from some apps, and can be subsequently generalised to other apps, which can be used to assist system-level of mobile testing. A 'motif pattern' may have multiple instances, which are referred as 'motif events' in this work. For example, in Figure 5.1, the demonstrated motif pattern is to 'click the menu icon and subsequently click one menu item'. The instances of this motif pattern are parameterised with specific locations of the menu icon and menu items, e.g., click the menu icon at the upper-left corner (M1) and the menu icon at the upper-right corner (M2). Note that although the three apps differ a lot in their functionalities with distinct categories ('Tools', 'Maps & Navigation' and 'Music & Audio', respective), they share the demonstrated same motif pattern, which can be used to generate tests that cover their core features.

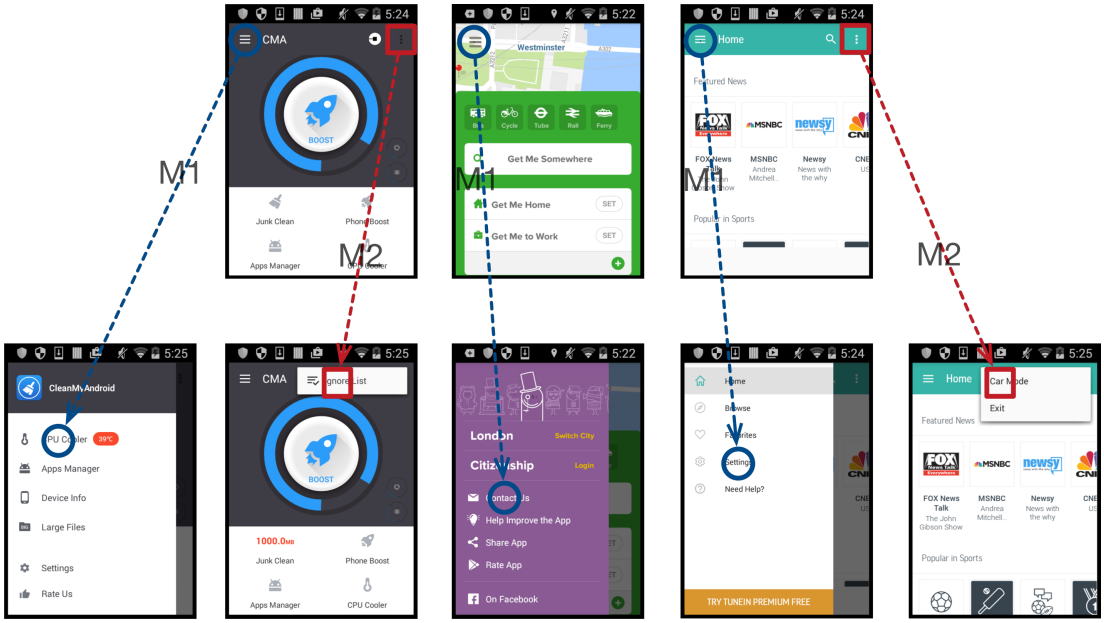



Figure 5.1: An example of crowdsourced motif events

The contributions of the work in this chapter are as follows:

- We propose the POLARIZ approach for harnessing crowd intelligence to support mobile testing. The approach uses a crowd testing platform which provides the remote mobile testing infrastructure and automatically records crowd generated event traces. It further mines the recorded traces and extracts crowd intelligence with its mobile interaction motif extraction algorithm, based on evolutionary computation. The learned motif events are subsequently used to enhance our previously proposed SAPIENZ approach, as a means of evaluating the effectiveness of POLARIZ for one of its potential downstream applications.

- Our novel crowd testing platform, also called as POLARIZ (available at: `http://mtest.uk`), which enables crowdsourced mobile testing from any source of app under test, via any platform, by any crowd. The platform mitigates the gap between automated mobile testing techniques and non-professional crowd testers, with features including AUT distribution, remote device control, permission control, crowd trace collection, test coverage measurement and crash detection.

- The first empirical study considering crowdsourcing for mobile test automation. We posted 1,350 tasks on Amazon Mechanical Turk to test 9 popular Google Play apps, each with at least 1 million user installs. We demonstrate the usefulness of

Polariz in harnessing crowd composes of members of the general public (with no necessary association of technical skills nor experience) to perform mobile testing tasks. We demonstrate the effectiveness of the automatically learned crowd motif events in enhancing the performance of existing automated mobile testing approaches (e.g., Sapienz) in terms of app activity coverage.

Section 5.2 of this chapter presents our Polariz approach, including its two major components: the Polariz platform and the Polariz motif extraction algorithm. Section 5.3 covers the implementation of our Polariz approach. Section 5.4 describes two empirical studies for evaluating the usefulness of our Polariz approach. The first study investigates its ability in harnessing the crowd to perform remote mobile testing and the second study examines the effectiveness of the crowd motif events learned by the motif extraction algorithm. Section 5.4.3 shows the results of the two empirical studies. Section 5.4.4 discusses the threats to validity of our evaluation and finally Section 5.5 summarises this chapter.

## 5.2 The Polariz Approach

The Polariz approach is designed to tackle the two main challenges involved in harnessing the non-professional crowd to perform remote mobile testing, and further to learn from crowd intelligence embodied in the crowdsourced manually constructed tests. The first challenge requires an intermediary platform able to harness a general public crowd to work on remote mobile testing tasks. The second challenge involves the representation and extraction of useful crowd intelligence.

A high-level workflow of Polariz is illustrated in Figure 5.2. Three actors are involved in the workflow:

1. App developer/researcher who has app testing requests and seeks improvement of their mobile test automation techniques;

2. Crowd workers/testers who have an interest in mobile testing or simply wish to get monetary reward;

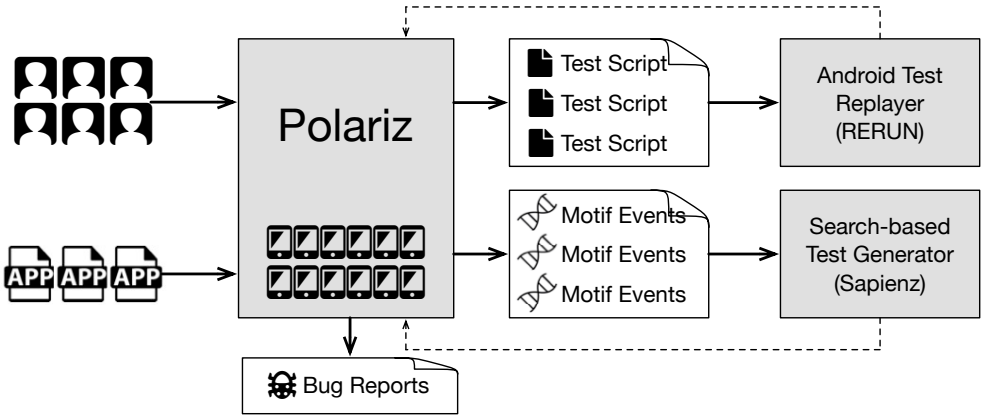



Figure 5.2: Overall workflow of Polariz

3. The intermediary platform (i.e., Polariz platform) where the crowd can work on the requester's mobile testing tasks, and where further automated processing of crowd interaction traces happens.

From the view of the intermediary Polariz platform, the inputs are the apps under test and also the crowd testers' interactions with these apps installed on Polariz' mobile device infrastructure. The outputs consist of three parts: First, the bug reports automatically generated during the crowd testing process; Second, the replicable test scripts generated based on crowdsourced test manual traces, which can be replayed via an Android test replayer (such as RERAN [131]); Third, the automatically summarised motif events learned from crowd interaction traces, which can be further used to enhance existing search-based mobile test generators (such as Sapienz). Both the Android test replayer and test generator can remotely connect to Polariz' mobile device infrastructure for test execution.

The two top level components of Polariz are its crowd testing platform (for manual trace collection), and its crowd motif extraction algorithm (for learning from crowd intelligence). We elaborate these two parts below.

### 5.2.1 The Polariz Platform

Detailed components of the Polariz platform are illustrated in Figure 5.3. Given a set of mobile apps under test, Polariz' subject dispatcher component will automatically

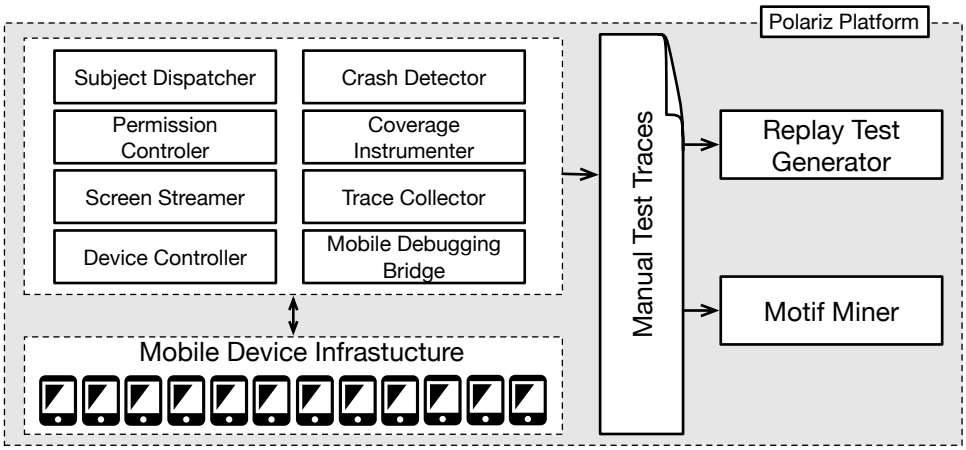



Figure 5.3: Detailed components of Polariz platform

instrument, assign and install each app on a real mobile device in its mobile device infrastructure. The screen streamer and device controller provide web services for controlling these real mobile devices. Crowd users can simply access the remote mobile devices with install apps via web browsers from any platform such as PC or their own Android or iOS mobile phones.

Exposing the access of hosted mobile devices to general public might raise security concerns, so Polariz has a permission control component that monitors crowd interactions, only permitting testing activities on the specified subjects.

During the the crowd testing process, Polariz' logging components such as a crash detector and trace collector automatically collect necessary information for generating crowd testing bug reports and manual test traces for further analysis.

### 5.2.2 The Polariz Crowd Motif Extraction Algorithm

We proposed the concept of 'motif event' for automated mobile testing in Chapter 3. A crowd motif event consists of a series of 'atomic event' (which cannot be further decomposed, e.g., press down a key) that plays some higher-level role. These motif events follow common user interaction patterns that may be frequently required in mobile app testing, in order to trigger app execution state transition. For example, fill a form and submit or drag an item and release. In this subsection, we present an algorithm for extracting these conservative motif patterns from crowdsourced manual

event traces.

Our idea of 'crowd motif' stems from DNA sequence motif. According to D'Haeseleer [98], a DNA sequence motif is a short, over-represented pattern with an assumed biological function. The crowd motif extraction problem is to find a set of recurring substrings within a set of strings, which can be described as follows:

Given a set of $N$ test event sequences $\mathcal{S} = \{S_1, \ldots, S_N\}$, each is formed by mobile interaction events from an event set:

$$Y = \{\text{Swipe}, \text{Rotate}, \text{Flip}, \text{Pinch}, \text{Click}_{\text{ROI1}}, \text{Click}_{\text{ROI2}}, \ldots, \text{Press}_{\text{key1}}, \text{Press}_{\text{key2}}, \ldots\}, \tag{5.1}$$

the crowd motif extraction problem is to find a set of instances $M = \{m_1, \ldots, m_n\} (n \leq N)$, where each $m_i$ is a $w$-sized subsequence of some sequences in $\mathcal{S}$, such that the information content of $M$ is maximised:

$$\text{IC}_M = \sum_{i=1}^{w} \sum_{y \in Y} p_{y,i} \log \frac{p_{y,i}}{\mathbb{B}_y}, \tag{5.2}$$

where $p_{y,i}$ is the probability of event $y$ at sequence position $i$, and $\mathbb{B}_y$ is the background distribution.

Since we model the crowd motif extraction problem according to the DNA sequence motif discovery problem (which has been widely studied in bioinformatics [93, 167, 190]). We reuse existing DNA motif discovery methods for our mobile testing scenario. We use a genetic algorithm to extract crowd motif events for mobile testing. The adapted crowd motif extraction algorithm is listed in Algorithm 5.1.

The algorithm extracts multiple motif patterns from a set of collected log-trace pairs. The log provides subject execution state information, such as transitions from one app activity to another, and the trace saves manual interactions that were used to trigger the app state changes. The log and trace items are linked via their timestamps. In order to learn from 'the wisdom of the crowd', Lines 2-3 extract the minimum trace that enables a transition from one activity to another. That is, there may exist many ways for user interactions to trigger the same app activity. We favour the 'minimum

**Algorithm 5.1:** Crowd Motif Extraction Algorithm

1 **Description:** Find $m$ motif patterns from $n$ subject log-trace pairs.
**Input**: A list of log-trace pairs $\mathcal{D} = [(L_1, T_1), (L_2, T_2), ..., (L_n, T_n)]$, where $L_i$ is the app execution state log and $T_i$ is the app event traces for the $i$th app; Number of motif patterns to find $m$.
**Output**: A list of recurring motif patterns $R = [r_1, r_2, ..., r_m]$.
2 $R \leftarrow []$, $S \leftarrow []$; ▷ initialisation
▷ get the minimum operations to switch from one activity to another
3 **for** each *(L, T)* in $\mathcal{D}$ **do**
4 $S \leftarrow S \cup getMinimumActivityTransitionTrace((L, T))$;
▷ find $m$ motif patterns by evolving candidate motif substring locations
5 **for** $i$ in $range(0, m)$ **do**
6 generation $g \leftarrow 0$;
▷ for each individual generate random candidate motif locations in S
7 $P \leftarrow initialisePopulation(S)$;
8 evaluate $P$ by calculating $IC_M$ for each individual in $Q$; ▷ see Equation 2
9 **while** $g < max\_generations$ **do**
10 $P' \leftarrow tournamentSelection(P)$;
11 $Q \leftarrow variation(P')$ ; ▷ crossover and mutate motif locations and length
12 evaluate $Q$ by calculating $IC_M$ for each individual in $Q$;
13 $Q \leftarrow elitismSelection(Q, P)$;
14 $g \leftarrow g + 1$;
15 $P \leftarrow Q$;
16 $r \leftarrow getBestIndividual(P)$; ▷ may contain zero or one motif location for $s \in S$
17 $R \leftarrow r \cup R$;
▷ excluding found motif substrings for next motif pattern
18 $S \leftarrow removeMotifSubstrings(S, r)$;
19 return $R$;

trace' that requires the fewest operations. The generated minimum transitional trace collection, $\mathcal{S}$, is represented as a list of strings, where each string is a minimum trace that triggers a distinct 'A to B' activity transition.

Lines 5-18 use genetic algorithm to find multiple motif patterns. At each iteration, it finds one motif pattern and excludes the matched motif substrings from $S$ (Line 18). The genetic algorithm evolves a population of individuals. Each individual represents a candidate motif pattern, which is a list of candidate motif locations in $S$. The individual fitness is evaluated based on the information content score, as described in Equation 2. The variation operator (Line 11) applies both crossover and mutation on the selected parent individuals. These two operators manipulate the location and length of each motif substring in the individual. The elite individuals (with highest information content score, i.e., their motif substrings are most conservative) are selected for the next generation. When the maximum allowed generation number is reached, the evolution stops and the algorithm saves the best individual found. The above process repeats until all $m$ motif patterns have been discovered.

## 5.3 Implementation

We have implemented the Polariz approach as a system also named Polariz. Our Polariz implementation consists of the two top level components as described in Section 5.2. The Polariz platform contains additional sub-components (shown in Figure 5.3).

Polariz' mobile device infrastructure consists of 9 Nexus-7 tablets, which are connected to a host PC via a USB hub. To enable manual trace record and replay, we adapt the Android 'getevent' tool for trace recording, and use the RERAN [131] tool for trace replay. For remote device control, we developed the Polariz platform based on the open sourced 'openstf' project [10], and we deployed the platform to a server in UK with a proxy service to speed up global visits.

We choose our Sapienz approach as the existing search-based approach that receives the learned crowd motifs. We use the Sapienz implementation obtained from our research artifact [248]. When improving Sapienz, we integrate the learned motifs into Sapienz' motifcore component, which combines the motif genes with atomic genes. In order to learn the crowd motifs from the collected event traces, we implement the Polariz motif extraction component in Python, according to Algorithm 5.1.

## 5.4 Empirical Studies

To investigate the usefulness of Polariz, we conduct two empirical studies: Study 1 investigates how effective the crowdsourced mobile testing enabled by Polariz is, with recruited non-professional testers from the general public. As the first empirical study on hosted remote crowdsourced mobile testing, there is no prior work against which to compare. Effectiveness is evaluated by examining participation level, participants' interest in performing the tasks, task execution speed and the coverage achieved by crowdsourced manual testing. Study 2 further evaluates the Polariz crowd motif extraction feature, by asking to what extend can the automatically learned crowd motif events improve our automated mobile testing approach, Sapienz.

Table 5.1: Nine popular Google Play subject apps ('#A' for number of activities; '#M' for number of methods; 'Installs' is measured in millions)

| Subject | Ver. | Category | #A | #M | Rating | Installs |
|---|---|---|---|---|---|---|
| HP All-in-One Printer Remote | 4.1.18 | Productivity | 74 | 13,616 | 4.1 | 10-50M |
| TuneIn Radio | 17.1 | Music & Audio | 27 | 13,474 | 4.4 | 100-500M |
| Trainline | 2.5.0 | Maps & Navigation | 41 | 7,497 | 4.3 | 1-5M |
| Power Security | 1.0.18 | Tools | 38 | 5,802 | 4.4 | 5-10M |
| Google Translate | 5.6.0 | Tools | 17 | 4,765 | 4.4 | 100-500M |
| Brightest Flashlight | 1.35 | Productivity | 27 | 6,087 | 4.3 | 5-10M |
| Duolingo | 3.39.1 | Education | 29 | 9,949 | 4.7 | 50-100M |
| Clean My Android | 1.1.9 | Productivity | 16 | 1,804 | 4.7 | 1-5M |
| Citymapper | 6.15 | Maps & Navigation | 32 | 25,998 | 4.5 | 1-5M |

### 5.4.1 Subject applications

We perform the empirical evaluation on 9 randomly selected real-world Google Play apps from top 500 most popular free/in-app-purchase apps as listed in the Google Play app store on December 20, 2016. We chose 9 subjects based on their availability of our hardware resources (9 Nexus-7 tablets in the Polariz device infrastructure). The 9 apps are closed-sourced and cover multiple app categories. Each app has at least 1 million user installs (according to Google Play). When we perform the random selection, we first exclude gaming apps that are not based on standard Android native UI components. To protect the crowd testers' privacy, we also exclude apps that request user account information after launching. The crowd was also notified that they should not disclose any personal information during the testing process. Detailed subject information including version numbers, sizes, ratings and the number of installs are presented in Table 5.1.

### 5.4.2 Experimental settings

We perform the two empirical studies on all 9 subjects as described above. For each subject, we assign the same app running environment, i.e., the same software and hardware configurations. These configurations mimic general real-world end-user testing scenarios, e.g., with real devices that have Google service framework and WIFI

network connection, but without providing app-specific contexts. For example, the 'HP All-in-One Printer Remote' app may require an HP printer for testing some of its functionalities. In our experiments, we do not provide such app-specific equipments for the generalisation purpose.

### Study 1: Harnessing Remote Crowd Testing

We need to recruit crowd workers and manage payments by using a third-party intermediary, which we achieved as described below.

**Crowd recruitment.** We use Amazon Mechanical Turk[1] (AMT) for recruiting non-professional crowd workers from the general public. AMT is currently one of the most popular crowdsourcing platforms for micro tasks with general crowd workers. We recruit AMT workers to perform remote mobile testing tasks on our Polariz platform by posting human intelligence tasks (HITs) on AMT. Anyone, from any country, who is eligible to work on AMT is allowed to work on our HIT assignments. We only disclose the task information and our Polariz web service URL via the AMT HIT for controlling the worker sources (i.e., only AMT workers are expected) because this may interfere the recruitment speed and Polariz' visitor statistics.

For motivating the crowd, we provide 1.5 USD payment for each approved submission, as the extrinsic incentive to the crowd workers. We expect each worker will spend 10 minutes or less on one HIT assignment. The payment rate is higher than current UK national minimum wage (7.2 GBP/hour) and also the US standard (7.25 USD/hour). Also, the intrinsic incentive consists of the opportunity for crowd workers to experience manual mobile testing, and maybe, also to test the remote apps for fun (we investigate the task interest level in the results section).

**Task design and quality control.** A clear task description is considered to be one of the most important factors for successful software crowdsourcing tasks [111]. This motivates our careful design of our HIT. The general workflow of our designed HIT task is as follows:

[1] `https://www.mturk.com`

1. The crowd worker views the task assignment description on AMT and can choose to accept or decline the task.

2. The worker follows the task instruction and works on our Polariz platform via any devices with a browser and performs manual testing on one arbitrarily selected app.

3. Upon finishing the testing task, the worker copies the Polariz generated app execution log as the proof of task completion and goes back to the AMT HIT.

4. The worker submits the automatically generated log and answers a questionnaire which contains 6 brief questions via AMT.

5. The worker waits for requester's review and gets paid via AMT, assuming their submission needs a 'sanity check' for appropriate -engagement.

In the task description, for comprehensive testing, we instruct the workers that the goal is to explore and trigger as many functionalities of the subject app as possible. A few detailed steps for accessing our Polariz platform are illustrated using snapshots. In the questionnaire, we ask 6 short questions to collect their feedback on the interest level of the task, and background information regarding the workers, including their daily mobile usage duration, software testing experience, country, gender and education level. No personal information that may reveal the work's personal identity is collected.

For quality control, we give three criteria to the workers, which form our 'sanity check' for task approval and consequent payment: First, the worker has tried to explore and trigger multiple app functions (preferably as many as possible). Second, the worker has tested the app for at least 3 minutes. Third, the submitted app execution log contains at least 300 lines. Normally these criteria can be easily satisfied by testing the app for a few minutes. We review each submission by checking above three criteria in a semi-automated manner. We measure these three criteria based on the submitted logs (for Criteria 1, we use at least two activities as the lower bound). As a further sanity check, we also manually inspect the submissions periodically. If a submission is rejected, we do not repost that assignment.

We posted 1,350 assignments from December 22, 2016 to January 2, 2017, in a continuous manner, in order to leave time to perform daily reviews. These 1,350 assignments were split into 150 HITs, each containing 9 assignments. All HITs and their assignments are the same. Each HIT may contain one or more task assignments. Each worker can work on multiple HITs, but can only work on one assignment in one HIT.

**Polariz deployment.** We deploy Polariz as a publicly accessible web service, at a server located in UK, plus a Linode cloud server as a proxy to speed up global visits. The mobile device infrastructure is hosted at the author's lab and connected to the front-end server. Accessing the remote device does not require authentication but the mobile devices is monitored and manipulated under Polariz' permission control component, where changes to environment settings are prohibited. The 9 subjects are pre-installed on 9 Nexus-7 tablets; one per device. User interactions are logged with timestamp information which can be mapped to the submitted logs. All subjects are reset to their initial states every half an hour. This is to avoid the case that one worker drives the app into a state that the subsequent workers cannot recover from. Of course, we can reset app state per worker, however we expect that multiple workers can collaboratively work one subject, by setting the reset duration to 30 minutes.

#### Study 2: Using Polariz to Learn Crowd Motifs

In Empirical Study 2, we learn crowd motifs by using Polariz' motif extraction algorithm. Subsequently, we integrate these motifs into Sapienz to investigate whether any improvement can be achieved.

**Performance metrics.** We believe that crowdsourced testing has inherent realism and domain-aware advantages compared to fully automated testing. The generated realistic tests may also have shorter lengths compared to alternative stochastic test generators. In order to investigate these claims for each objective we need to carefully design controlled experiments. As the first study on crowd-generated tests, in this work we focus on the coverage objective, which is an important fundamental metric for software testing [260, 279, 407]. In terms of granularity of coverage, we measure app activity coverage since more fine-grained measurement requires reverse engineering

of the subjects, which is prohibited for many commercial closed-sourced apps. App activity coverage has been adopted in previous studies on automated mobile testing [85, 248]. This metric thus gives us a baseline to assess and compare the ability of tests to 'explore' the AUT.

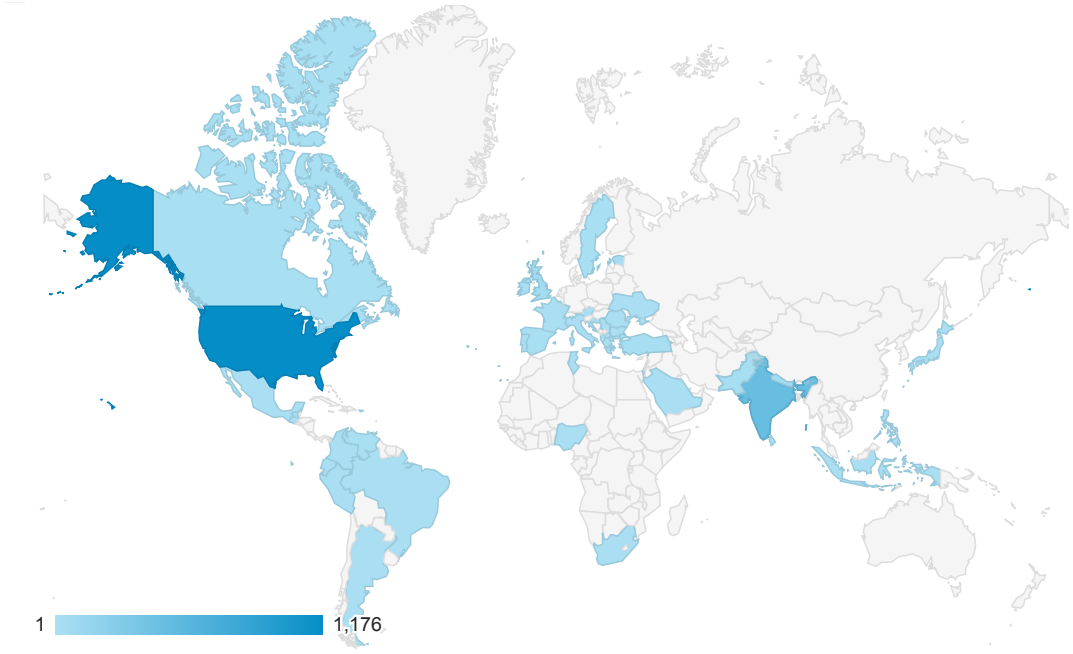
**Motif extraction.** As explained in Section 2.2.6, we learn generalised event patterns because high-level events learned on one subject itself is intuitive and has already been proved useful in a previous study [103]. In our experiment, we perform a leave-one-out evaluation on the extracted crowd motifs. That is, when evaluating a subject, we will use only the motifs extracted from the remaining 8 subjects' event traces. For each subject, we apply the Polariz motif extraction algorithm to learn three motif patterns.

**Improving Sapienz.** To examine whether the learned generalised motifs are helpful in improving app activity coverage, we run Sapienz without any motif information and compare results to these obtained from running Sapienz with the learned crowd motifs. On each subject, we run Sapienz for 60 minutes wall-clock time. We set the delay between each two events to 200 ms so that, given the same amount of wall-clock time, roughly the same number of events will be used. This setting aims for a fair comparison between the two Sapienz versions with and without motif genes. For Sapienz parameters, we use the default settings as used in the original paper. In all experiments, the parameters were not tuned, to avoid any implicit experimental biases that might otherwise arise.

We run all experiments on the same MacBook Pro with 2.3GHz Intel Core i7 CPU and 16G RAM. The mobile side for app execution is a Nexus-7 real device.

### 5.4.3 Results

This section presents the evaluation results of the two empirical studies. Section 5.4.3 shows the connectivity, participation, task interest level and crowd performance of the remote crowdsourced mobile testing enabled by Polairz. Section 5.4.3 examines the effectiveness of the learned crowd motifs in improving the Sapienz approach, in terms of app activity coverage.




Primary Dimension: **Country** City Continent Sub Continent

| Country | Sessions | Sessions | contribution to total: Sessions |
|---|---|---|---|
| | 1,931<br>% of Total: 100.00% (1,931) | 1,931<br>% of Total: 100.00% (1,931) | |
| 1. United States | **1,176** | 60.90% | |
| 2. India | **481** | 24.91% | |
| 3. United Kingdom | **53** | 2.74% | |
| 4. Nepal | **44** | 2.28% | |
| 5. Estonia | **28** | 1.45% | |
| 6. Pakistan | **19** | 0.98% | |
| 7. Turkey | **17** | 0.88% | |
| 8. (not set) | **11** | 0.57% | |
| 9. Ireland | **10** | 0.52% | |
| 10. Peru | **9** | 0.47% | |

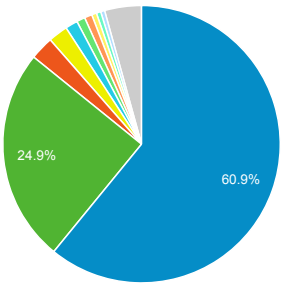



Figure 5.4: Visitor statistics of Polariz service according to Google Analytics

Table 5.2: Global connectivity test of Polariz remote crowd testing service (service delay measured in milliseconds)

| Location | Requests | Min | Max | Avg | Std Dev | Loss |
|---|---|---|---|---|---|---|
| New York | 40 | 70.103 | 70.544 | 70.282 | 0.323 | 0% |
| Miami | 40 | 120.323 | 120.527 | 120.453 | 0.355 | 0% |
| Dallas | 40 | 112.626 | 112.876 | 112.727 | 0.096 | 0% |
| San Francisco | 40 | 158.458 | 158.670 | 158.555 | 0.408 | 0% |
| Seattle | 40 | 130.806 | 131.260 | 130.976 | 0.474 | 0% |
| Toronto | 40 | 85.750 | 86.392 | 86.122 | 0.318 | 0% |
| Frankfurt | 40 | 17.785 | 17.872 | 17.834 | 0.033 | 0% |
| London | 40 | 0.361 | 0.584 | 0.465 | 0.090 | 0% |
| Paris | 40 | 15.370 | 15.452 | 15.421 | 0.093 | 0% |
| Amsterdam | 40 | 9.943 | 10.113 | 10.026 | 0.063 | 0% |
| Sao Paulo | 40 | 195.777 | 195.807 | 195.795 | 0.442 | 0% |
| Singapore | 40 | 186.548 | 186.709 | 186.598 | 0.065 | 0% |
| Sydney | 40 | 295.810 | 296.022 | 295.921 | 0.084 | 0% |
| Tokyo | 40 | 246.735 | 249.681 | 248.273 | 1.161 | 0% |

**Study 1: Harnessing remote crowd testing**

In general, Polariz successfully assisted the crowd workers to complete all 1,350 task assignments posted on AMT, from December 22, 2016 to January 2, 2017. In the following, we show the evaluation results on detailed aspects.

**Connectivity.** One of the advantage of crowdsourcing is its ability to recruit from a large pool of global works. To leverage this property we are interested in the service connectivity of the Polariz service. As shown in Figure 5.4, according to the visitor tracking data from Google Analytics, from December 22, 2016 to January 2, 2017, there were 1,931 sessions of visits to our remote crowd testing service. Of these sessions, 56.9% come from new visitors and 43.1% from returning visitors. The records show the traffic comes from at least 9 countries, with most are from the USA (60.90%) and India (24.91%). Note that there are other countries with large populations (such as China) whose workers are ineligible to work on AMT, so there are no visits from these countries. Table 5.2 presents the ping test[2] results for the Polariz remote crowd testing service, from 14 global sites. The average response times range from 0.465 ms (London) to 295.921 ms (Sydney). This result indicates a reasonably good connectivity of our service for performing remote crowdsourced mobile testing.

[2]https://tools.keycdn.com/ping

**Participation.** During the 12 days of experimentation time, our 1,350 posted HIT assignments have all been finished by the crowd workers. Of all submitted solutions, 1,075 (79.6% ) were approved, according to the criteria for quality control discussed in Section 5.4.2. We received 1,350 submissions from 434 distinct workers. Results from our questionnaire show that these workers come from 24 countries, while 99% submissions are from the top 10 most frequently submitting countries (as listed in Figure 5.5). Note that the number of countries is inconsistent with the traffic we observed according to Google Analytics; the questionnaire reveals for wider country participation than that would be suggested by the Google Analytics data. Our interpretation is that a small number of AMT workers may use proxies to visit the AMT (as the service is disabled in their countries), or they may misunderstand the question and filled in their nationalities instead of their country of residence.

Figure 5.5 presents worker demographic information based on the 1,350 responses submitted by the crowd. Most workers come from USA (76.7%) and India (14.1%). More male workers (64.4%) submitted than female workers (31.4%). Regarding the educational level, 88.3% workers at least attended some college education (including undergraduate students). This generally high educational level is consistent with previous studies [326, 327], although our results show that there are more workers with some college education than those holding a Bachelor's degree.

Since our remote testing tasks requires basic skills for interacting with mobile apps, we expect the crowd to have a certain amount of daily mobile usage. Our questionnaire results on 'Daily Mobile Usage' suggest that only 0.9% of the respondents spend less than 1 hour per day on mobile usage, indicating that our expectation is reasonable. We also recruit the crowd from the general public rather than software testing experts. As the distribution demonstrated in Figure 5.5, 72.3% respondents have less than one year testing experience, and the remaining 27.7% have at least one year's experience in software testing. Given that we recruit from the general public, a proportion of over a quarter having testing experience was a surprise to us (since testers do not occupy 1/4 of the world's population). Our understanding is that their experience may come from working on testing tasks posted on AMT or other crowd testing platforms such as uTest or they may have professional career experience in software testing. Furthermore,

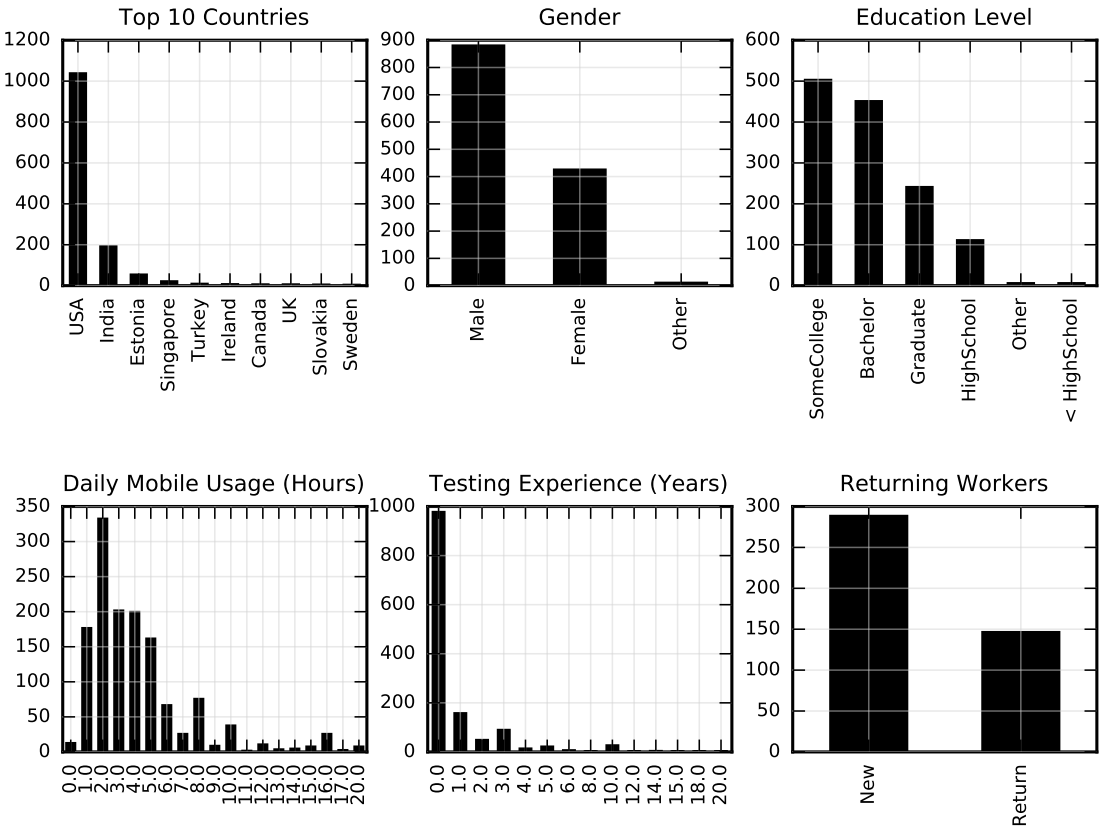



Figure 5.5: Worker demographic information based on 1,350 submitted assignments

those with testing experience may favour our HIT, while those without such experience may have self-selected out. Lastly, 66.4% workers are 'new', i.e., they only completed one task, while the remaining workers completed at least two tasks. This high rate of returning workers may be correlated with the interest level of our task: a topic to which we now turn.

**Interest level.** Gamification has been widely studied in software engineering as a means of increasing the engagement of workers [310]. One might regard our remote crowdsourced mobile testing as a naive form of 'gamification' on mobile testing, i.e., to explore more app functionalities and get paid for so doing. Figure 5.6 shows the feedback from the crowd on the interest levels of our task. The boxplot suggests a mean rating of 2.3 (between 2-'Interesting' and 3-'Normal'), and a median rating of 2. This relatively high rating of interest level may explain the high rate of returning workers revealed in the results of Figure 5.5. A detailed distribution of the number of submitted tasks by each worker is given in Figure 5.7. The distribution shows that, although there is a high rate of returning workers, the total submissions are not dominated by a small

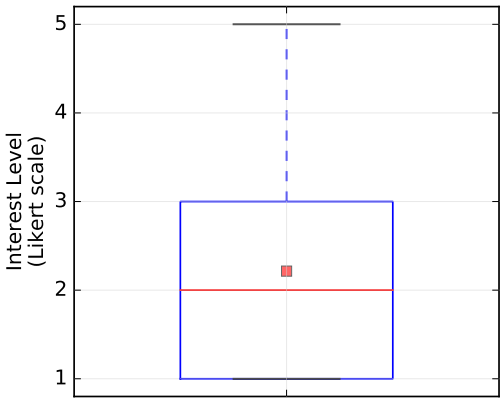



Figure 5.6: Worker feedback on self-assessed interest level of the task (1 = Very Interesting; 5 = Very Boring)

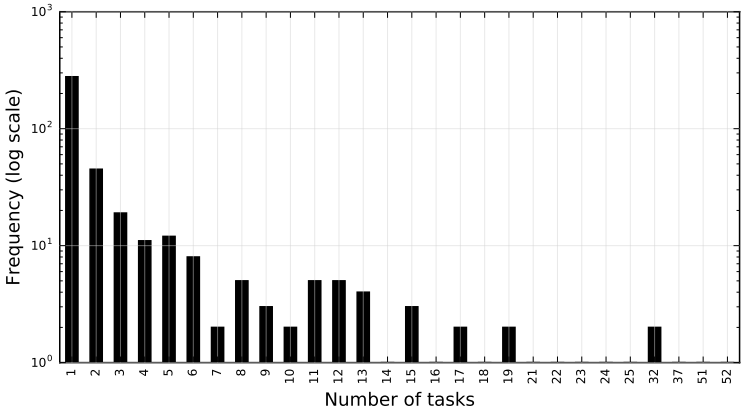



Figure 5.7: Distribution of number of submitted tasks per worker

number of 'super workers'.

**Crowd performance.** We investigated crowd performance along two dimensions, the speed of task performance and the thoroughness of crowdsourced manual testing in terms of activity coverage. The speed data is extracted from AMT task records and also the app execution logs submitted by the crowd. The app activity coverage data is calculated based on the submitted logs, which is produced by Android Logcat. The log information contains detailed activity launch, warning and error information.

In practice, many testing scenarios may be sensitive to test speed. Figure 5.8 presents three boxplots on the crowdsourced mobile testing enabled by Polariz. The 'Create-Accept' time is the elapsed time from posting a task on AMT to a worker consent/accept

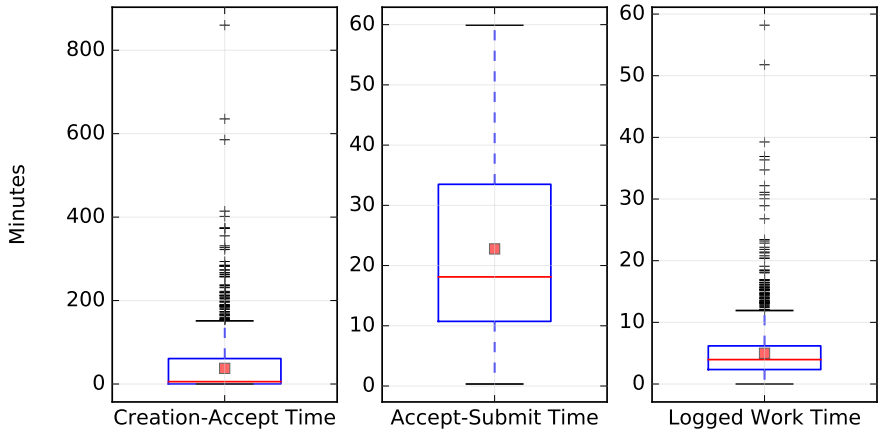



Figure 5.8: Crowd performance data: speed of task acceptance and completion

to work on the task. In the first boxplot, the time for the 75th percentiles is 61.0 minutes and 73.3% of the posted tasks were accepted within one hour. The 'Accept-Submit' time reports the time from task acceptance to submission of a solution by the crowd worker. The second boxplot reveals that all 1,350 posted tasks finished within one hour, with a median value of 18.1 minutes. Note that this time cost may not reflect the actual working time, because the worker may simply accept the task, and work on something else first. Thus we regard the data presented in the second boxplot as an upper bound on the working time. To further examine the lower bound, we check the crowd's working time based on the logs submitted. The logged time may not reflect the time required to become familiar with our Polariz platform, thus we regard it as a lower bound. As shown in the third boxplot in Figure 5.8, the IQR (25th to 75th percentiles) area shows a range of 2.3 to 6.2 minutes, which falls into our expectation on the working time, i.e., within 10 minutes, as we have mentioned in Section 5.4.2.

The crowd's performance in terms of test coverage is shown in Figures 5.9 to 5.11. First we examine the overall coverage and then consider the detailed coverage results for each of the subjects.

Figure 5.9 shows boxplots that depict the number of covered unique and non-unique activities per task. For non-unique activities, the number of triggered activities are 7 to 23 for IQR, while the number for unique activities is 3 to 9. Considering the real-world complexity of the subjects, and our testing tasks are designed to be lightweight/micro

tasks, this coverage performance is reasonable and is within our expectation.

The idea of crowdsourcing is to leverage the collective intelligence of the crowd. We are thus interested to report the collective coverage achieved by the crowd. Figure 5.10 shows the cumulative coverage on all 9 subjects. The tasks used in the x-axis are sorted in ascending order, by their submission time. In total, 21,440 non-unique activities have been manipulated by the crowd. In terms of coverage, 182 out of 301 total unique activities have been covered on 9 subjects. This yields a percentage 60.5% activity coverage.

A detailed analysis of the coverage achieved on each of the 9 subjects, is presented in Figure 5.11. Note that the subject assigned to the crowd is randomly chosen, thus the x-axis on the number of tasks for each subject may vary slightly, but each subject corresponds to at least 100 tasks. In all 9 cases, the cumulative coverage grows rapidly for the first 10 tasks and subsequently 'plateaus out'. In a few cases like the 'TheTrainline' subject, the coverage was still able to grow after more than 100 tasks have been considered.

The highest coverage is achieved on the 'CleanMyAndroid' subject (87.5%). While the lowest coverage is on the 'All-in-One Printer Remote' subject (21.6%), which is the only subject with a coverage below 60%. This low coverage is caused by the app-specific contexts which require external hardware to be present, such as connecting to a HP printer. In our experiments, such external hardware was unavailable.

**Study 2: Learning crowd motifs**

This section presents our evaluation of the ability of Polariz to learn crowd motif genes that can improve search-based mobile test automation. Our results were based on the collected crowdsourced event traces in Study 1. As a baseline for comparison, we first report Sapienz' performance on the subject dataset without any motif genes. We also provide a group of Venn diagrams to show the differences between the coverages achieved by Sapienz and the crowd respectively. Subsequently, we apply our Polariz crowd motif extraction algorithm to the collected traces and integrate the learned

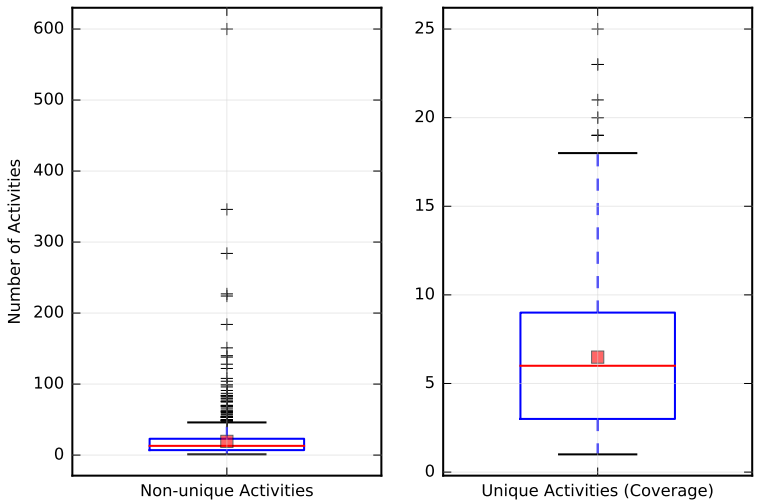



Figure 5.9: Number of covered activities per task

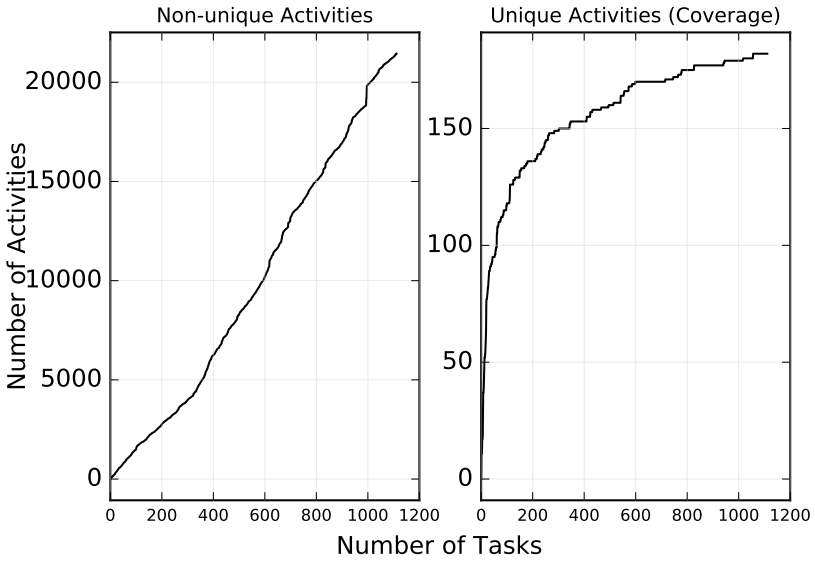



Figure 5.10: Overall cumulative coverage

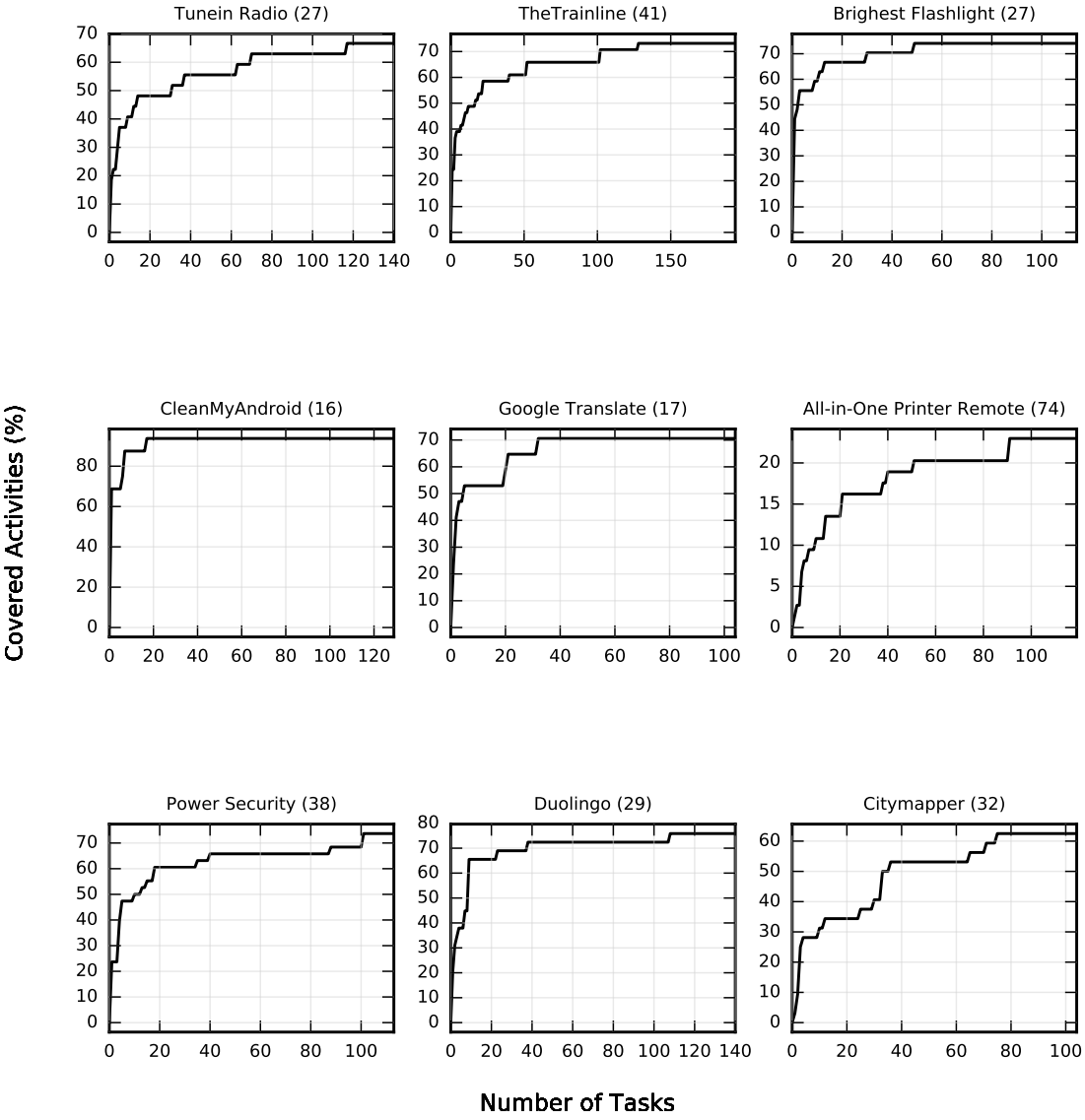



Figure 5.11: Coverage by subject, achieved by the crowd. The number in the bracket shows the total number of activities for that subject.

crowd motifs into Sapienz, reporting any improvements or reductions in effectiveness that accrue.

Figure 5.12 shows the coverage achieved by Sapienz without motif genes. When normalised, the coverage on the 9 subjects ranges from 5.4% (All-in-One Printer Remote) to 82.4% (Google Translate), and the mean coverage is 32.6% (98 out of 301 activities). This mean coverage is much lower compared to the mean coverage (60.5%) achieved by the crowd (which consists of 434 human workers). Note that we are not claiming a fair comparison between fully automated testing and crowdsourced manual testing, as it is inherently impossible to conduct a fair comparison between human and machine. However, it is useful to understand the degree of complementarity between human and machine.

We map the coverage for each subject to the coverage achieved by the crowd, as the Venn diagrams illustrated in Figure 5.13. From the 9 Venn diagrams we can see that the crowd generally covered more app activities than the fully automated Sapienz approach. However, they complement each other on 4 out of the 9 subjects. There is one case, i.e., on the 'Google Translate' subject, Sapienz triggered more activities than the crowd.

Finally, we analyse the effectiveness of the crowd motifs learned by our Polariz motif extraction algorithm. In Figure 5.14, we draw the coverage achieved by both Sapienz with and without the learned motif events, where the blue (darker grayscale) lines indicate the performance of Sapienz with motifs, and the red (lighter grayscale) lines denote results for Sapienz without a motif. As suggested by the line charts of cumulative coverage on each of the subjects, the learned motifs were able to enhance Sapienz in achieving higher test coverage in 6 out of 9 cases.

In the remaining 3 cases, the integrated motif genes led to neither improvement nor disimprovement in terms of app activity coverage. Recall that the motifs were learned on the traces excluding those from the subject itself, we believe that these learned motifs are generalised and attribute to the crowd intelligence underlying the crowdsourced event traces. Also, parameters used in the experiments have not been tuned. Therefore, our results are the fair lower bounds on the improvement that could be expected to

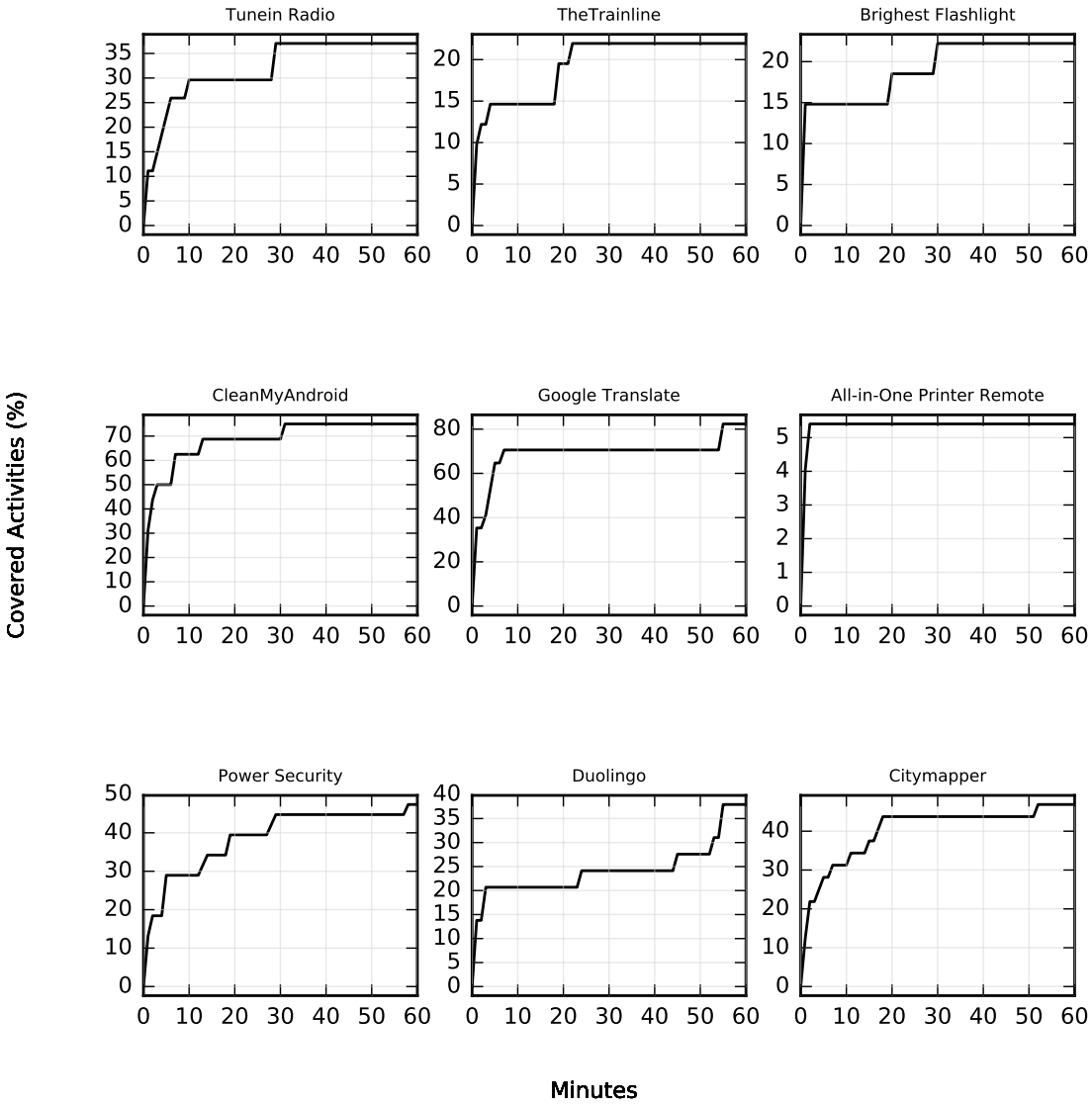



Figure 5.12: Coverage by subject, achieved by Sapienz without motif genes

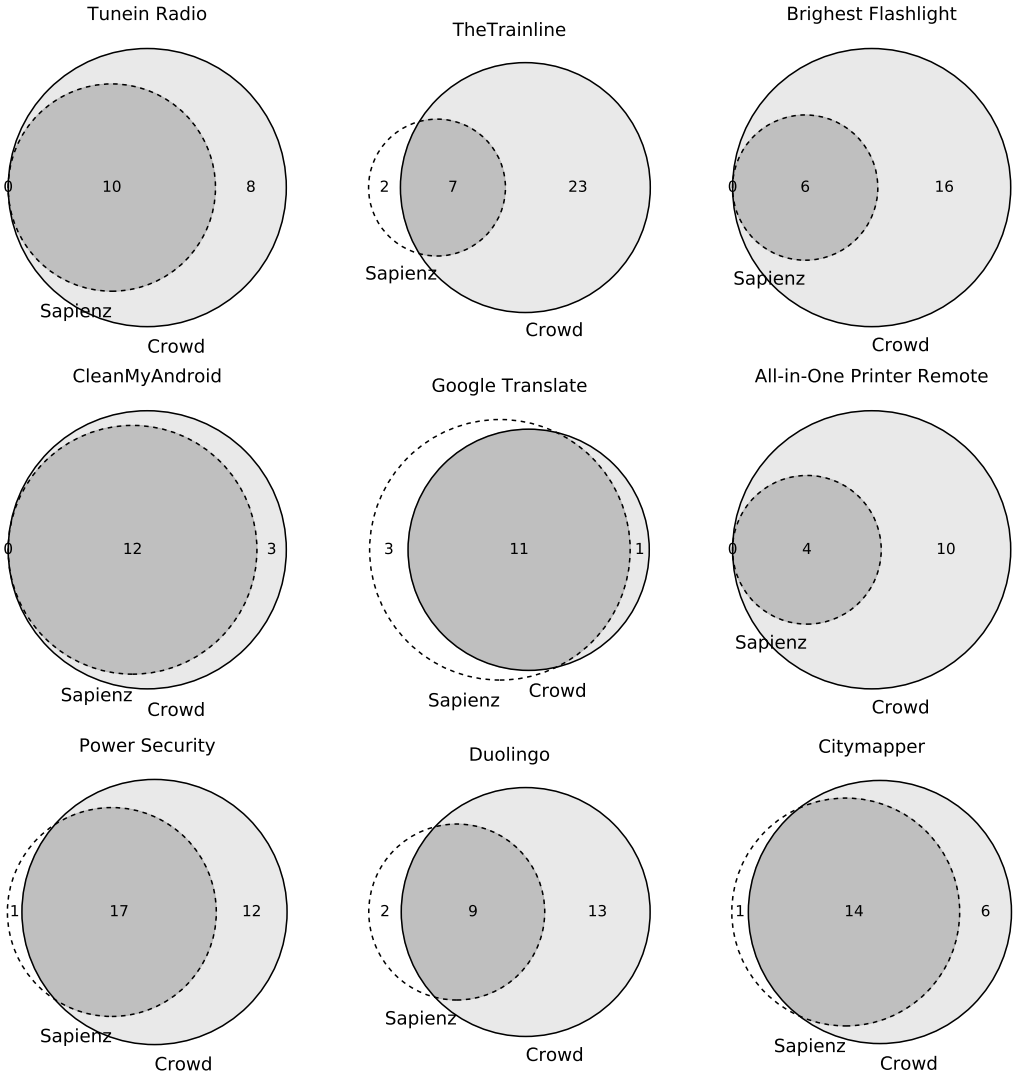



Figure 5.13: Venn diagrams of covered activities by Sapienz and the crowd

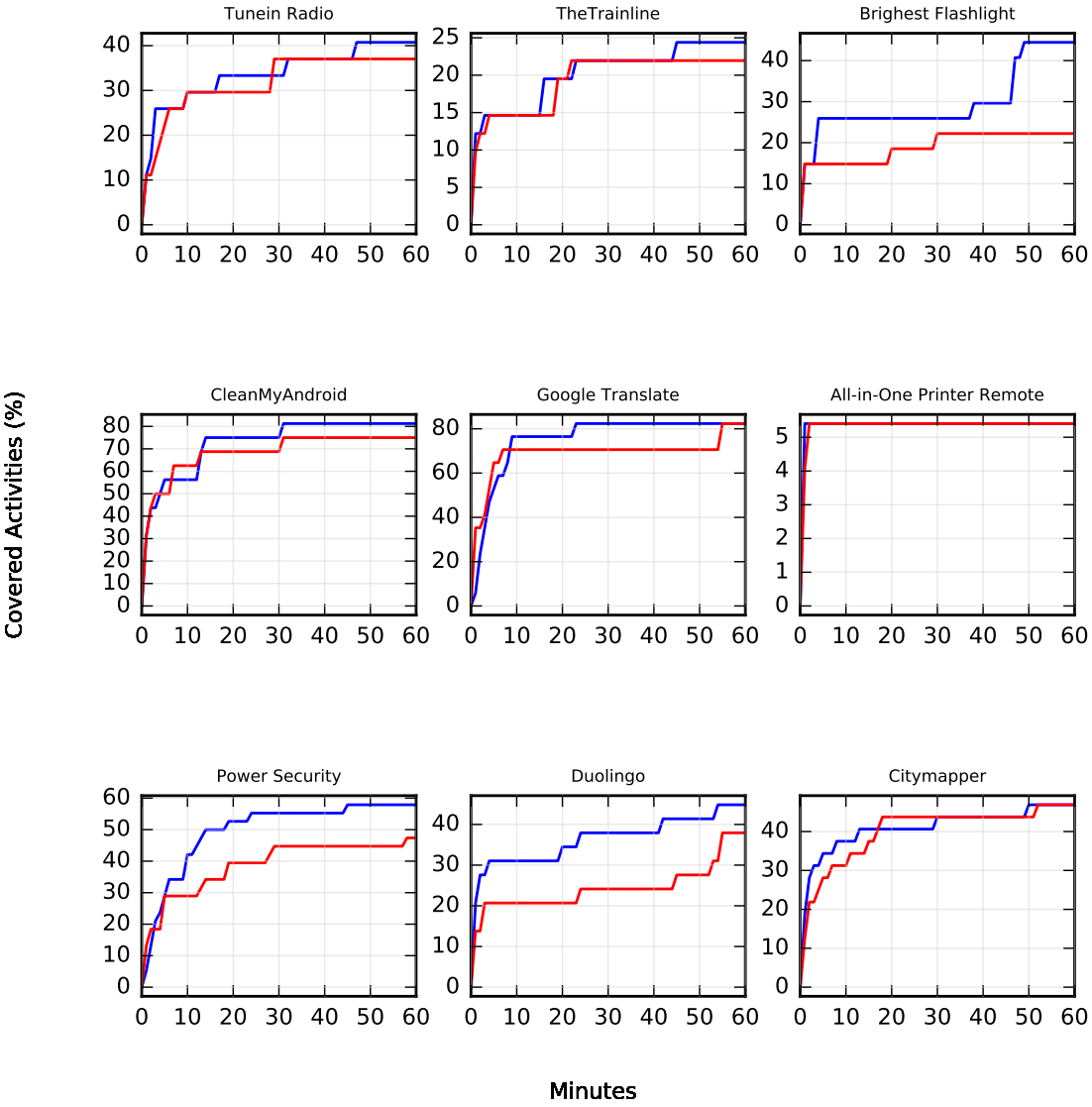



Figure 5.14: Coverage achieved by Sapienz without motif genes, and Sapienz with motif genes learned by Polariz. Lower (red/lighter grey) lines denote Sapienz without motif genes.

accrue; parameters tuning and more 'targeted' learning might improve the results we report here.

### 5.4.4 Threats to Validity

In this work, we have studied the effectiveness of our Polariz approach for harnessing a general public crowd to perform crowdsourced mobile testing, and subsequently in mining the crowdsourced data to support our Sapienz approach to automate test input generation. The primary threat to validity of our empirical studies is the threat

to external validity. Our subject dataset excluded two types of mobile apps, i.e., games (with non-standard Android UI components) and those have an initial login activity (for protecting the crowd's privacy). Thus our study results may not generalise to these apps. To partly mitigate the generalisation issue, we randomly chose apps which fall into multiple app categories from widely-installed real-world apps, each of which has at least 1 million installs.

The threats to internal validity lie in the way that our empirical studies were conducted. To minimise these threats, we tested both the components of Polariz and the scripts for data collection and analysis. One threat to internal validity that we cannot avoid is related to the permission control component of Polariz platform: To guarantee that the app testing contexts (e.g., WIFI connection) will not be changed by the crowd workers, the permission component disallows any call to external activities that are not part of the subjects. It is possible that for certain subjects, such calls to external activities are a precondition to trigger some of their own activities. This may lead to lower coverage performance for both the crowd and Sapienz. However, the findings for the relative effectiveness of the learned crowd motifs compared to the baseline should remain robust because the experiments were conducted under the same environment (including permission control in both cases). Of course, we cannot discount the possibility that such security sensitive blocking might have disproportionately affected one or other of our two treatments.

## 5.5 Summary

This chapter has presented the Polariz approach for harnessing a non-professional crowd to perform remote mobile testing tasks, and subsequently to enhance other search-based mobile test automation approaches. The approach uses a platform with a mobile device infrastructure, remote device control and screen streaming, automated subject distribution, permission control and crowd trace collection. With this approach, a non-professional crowd from the general public (such as those from Amazon Mechanical Turk) can contribute to mobile testing from anywhere with any clients with a web browser (e.g., desktop PC, Android, iOS or Windows Phone mobile devices). As part

of the Polariz approach, it automatically learns from the collected crowd event traces and models the underlying crowd intelligence as motif patterns, seeking to provide useful motif events for enhancing existing test automation techniques such as Sapienz.

Our evaluation results on 9 popular Google Play apps show that Polariz was able to harness 434 crowd workers from 24 countries to perform 1,350 task assignments. The automatically learned motif genes improved Sapienz' performance in terms of activity coverage, on 6 out of 9 subjects. On the remaining 3 subjects, the performance was no worse. We also found that our fully-automated Sapienz (without human knowledge) and the general public crowd complemented each other in testing 5 out of 9 subject apps.

# Chapter 6

# Future Work

Our results reported in this thesis are for automated machine-to-machine testing. Our work highlighted a need to consider robotic testing. This remains largely a topic for future work, but in this chapter we map out a possible vision by providing an initial proof of concept for robotic testing.

Most existing automated mobile testing techniques (including our Sapienz, Ocotpuz and Polariz) perform intrusive testing which requires modifications on the mobile device/app under test. The generated test interactions are based on simulated event signals sent into the mobile operating system. These simulated events may be unrealistic to real users (e.g., simultaneous clicking using more than five fingers). Thus these techniques require developer permissions and assume the simulated interactions can actually simulate real-world physical interactions triggered by end-users. These assumptions do not always hold in various scenarios in real-world mobile testing. We focus on the realism objective of the generated tests for future work. One possible way to tackle the test realism problem is to use robotic testing to mimic end-user testing that has happened in the real-world. Robots are widely used for many repetitive tasks. Why not mobile testing? Robotic testing could give testers a completely new form of 'blackbox' testing, that is inherently *more* black box than anything witnessed previously.

In the following section, we provide a preview of a proof-of-concept robotic mobile

testing system, Axiz, for truly blackbox automation: We first provide a comparison between the state-of-the-art simulation-based mobile test automation and our proposed robotic mobile testing. We give the scenarios for which robotic testing is beneficial (even essential), introducing a robotic mobile device test generator, Axiz. We illustrate the application of Axiz to a popular Google Calculator app.

## 6.1 A Manifesto for Robotic Testing

We believe that handheld devices require a rethink of what it means to be black box when testing. The user experience of handheld devices is so different to that for desktop applications, that existing 'machine-to-machine' black box test generation lacks the realism, usage-context sensitivity and cross-platform flexibility needed to quickly and cheaply generate actionable test cases.

This section sets out a manifesto for Robotic Testing, in which the execution of the generated test cases is performed in a truly black box (entirely non-intrusive) manner, using the cyber physical interface of the device, rather than machine-to-machine communication between test two and up on the test[1]. Table 6.1 compares manual, robotic and traditional automated testing techniques.

**Realism:** For Android testing, *MonkeyLab* [234] generates test cases based on the app usage data. There are also several published approaches to realistic automated test input generation for web-based systems [60]. Nevertheless, there is little or no attention to realism. A test sequence that reveals a crash, will not be acted upon by a developer who believes the test sequence to be unrealistic. All automated test data generation may suffer from unrealistic tests, because due to inadequate domain knowledge. However, there is an additional problem for the mobile paradigm: the tests may be simply unachievable by human, for example requiring simultaneous clicking using more than five fingers.

[1]The authors would like to thank Andreas Zeller, for his invited talk at the 36th CREST Open Workshop (COW 36: crest.cs.ucl.ac.uk/cow/36/slides/COW36_Zeller.pdf), at which he gave a playful video of a disembodied synthetic human hand, automatically interacting with a mobile device. This was one of the inspirations for our proposal.

Table 6.1: An overview of the criteria to consider when choosing from manual, simulation-based and robotic-based testing approaches

| **Aspects** | **Considerations** | **Manual Testing** | **Automated Testing** | | |
|---|---|---|---|---|---|
| | | | **Simulated Interaction** | | **Robotic-based** |
| | | | **Emulator-based** | **Device-based** | |
| Target | Test apps | Yes | Yes | Yes | Yes |
| | Test devices | Limited | No | Limited | Yes |
| Dependency | Platform support | Cross-platform | Platform-dependent | Platform-dependent | Cross-platform |
| | Platform version | Independent | Dependent | Dependent | Independent |
| Integrity | Modify OS | Not needed | In most cases | In most cases | Not needed |
| Permission | Developer privilege | Not needed | Needed | Needed | Not needed |
| Cost | Cost | High | Very low | Low | Medium |
| Scalibility | Scalibility | Very low | Very high | High | Medium |
| Interaction | Realism | High | Very low | Low | Medium |
| | Complexity | Moderate | Very Complex | Very Complex | Complex |
| | Sensor activation | Uncontrolled | No | No | Controlled |
| Performance | Accuracy | Vary | Very high | Very high | High |
| | Speed | Slow | Fast | Fast | Moderate |
| | Reliability | Low | High | High | High |
| User Experience | Test imaging | Limited | No | Yes | Yes |
| | Test touch screen | Limited | No | No | Yes |
| | Test IMU, NFC sensors | Limited | No | No | Yes |
| | Test UI response | Limited | Limited | Yes | Yes |
| Functionality | Oracle | Human | Automated | Automated | Automated |
| | Internal States | Not accessible | Accessible | Accessible | Not accessible |
| | Test LBS | Yes | No | No | Yes |
| Compatibility | Hardware | Yes | No | Yes | Yes |
| | Platform | Yes | Limited | Yes | Yes |
| | Network | Yes | No | Yes | Yes |

**Device Independence:** Existing white box and (claimed) blackbox automated testing require modification of the behaviour of either the app under test or the platform or both. Even techniques that are regarded as black box, communicate with the app though simulated signals rather than those triggered via real sensors (e.g., touch-screen, gravity-sensor) on the mobile device.

Robotic Testing uses the same cyber-physical interface as the human user, it is also less vulnerable to changes in the underlying platform, API interfaces and implementation details. In a world where time-to-market is critical, the ability to quickly deploy on different platforms is a considerable advantage.

**Cost-Benefit:** Human based testing is considerably expensive, yet it enjoys a great deal of realism and device independence. By contrast, existing automated test data generation is relatively inexpensive, relying only on computational time, yet it lacks realism and device independence. Robotic Testing seeks the best ratio of cost-to-benefit, and combines the best aspects of human-based testing and existing machine-to-machine automated testing.

Although robotic technology has historically proved to be expensive, we are currently witnessing a rapid decrease in the cost of robotic technology. Crowdsourcing, too, is currently reducing the cost of human-based testing [247], yet it seems unlikely that crowdsourcing would prove to be ultimately cheaper than Robotic Testing.

**Reduced Reliance on Assumptions:** Traditional automated test techniques make a number of assumptions about the system under test, whereas human-based test data generation relies on fewer assumptions. Robotic Testing is much closer to human-based testing in the number of assumptions made, yet its ability to generate large numbers of test cases cheaply is much closer to existing automated testing.

## 6.2 The Axiz Framework

Our Axiz Robotic Testing system architecture is depicted in Figure 6.1. The framework contains two high-level components: the 'robotic test generator' (for generating realistic

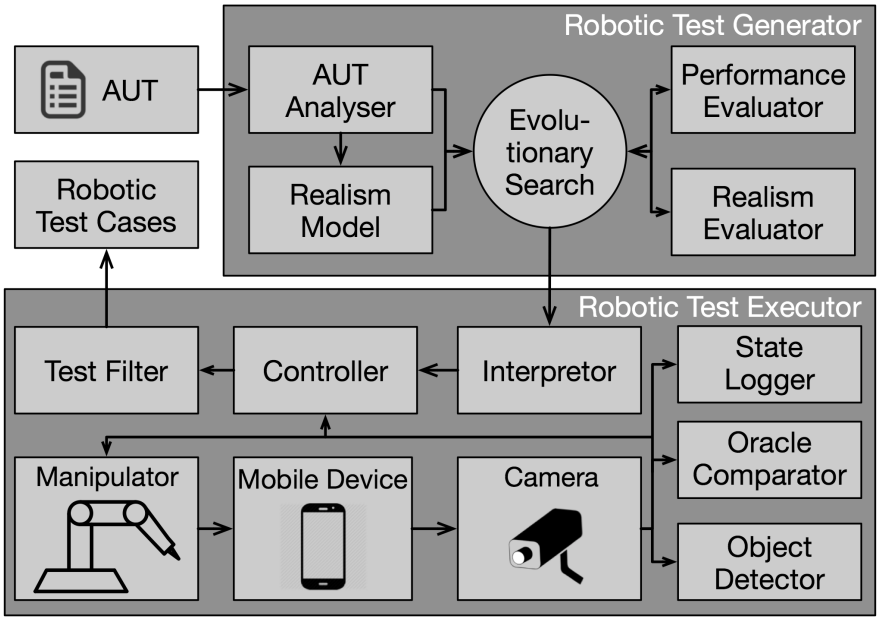



Figure 6.1: The framework of the Axiz robotic mobile testing system

test cases), and the 'robotic test executor' (for further execution and filtering of the tests). The filtering stage removes tests that can be detected to be unexecutable in a real-world setting.

**Robotic test generator.** The robotic test generator starts by analysing the application under test (AUT). The extracted information of the AUT (including app categories, static strings and APIs, etc.) is used to adjust a 'realism model'. The realism model uses previously-collected empirical data containing known-realistic test cases.

Based on a series of observations of human usage, we compute a comprehensive list of properties (e.g., the delay between two adjacent events, event types and event patterns), to capture the characteristics and properties of the underlying the real-world test cases. The hope is that these characteristics will capture what it is to be 'realistic', so that these can be used to guide and constrain automated test data generation.

The realism model, together with the AUT, are passed into the evolutionary search component for generating and evolving test cases. The source of 'realism' for the individuals being evolved, derives from two aspects of our approach: Firstly, by reusing and extending realistic test cases (e.g., Robotium or Appium test scripts), we draw on previous tests manually written by the app testers. Secondly, by searching a solution space constrained by the realism model, we constrain our search to generate test cases

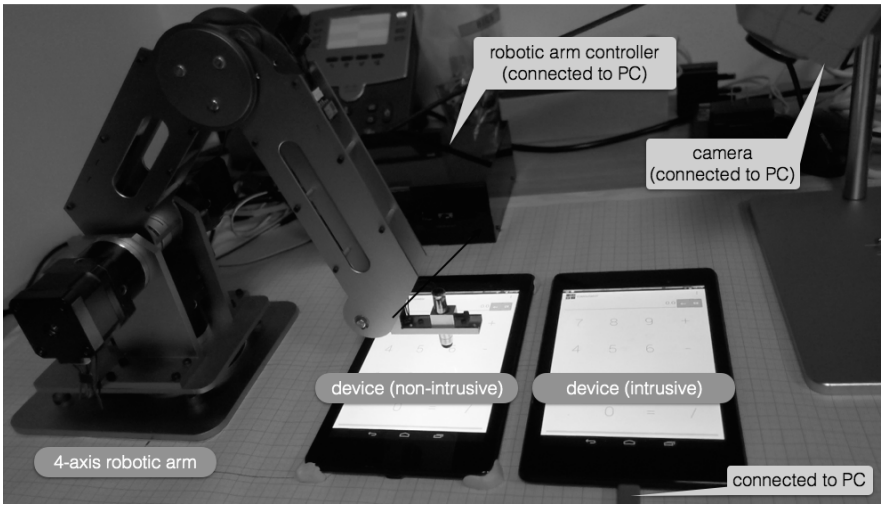



Figure 6.2: Testing mobile apps with a 4-axis robotic arm

that meet the constraints identified earlier from crowdsourced tests.

The fitness of the generated test cases is evaluated based on their performance (such as code coverage and fault revelation) and realism as assessed by the realism model.

**Robotic test executor.** The generated test case candidates are further validated by executing them on a physical device, thereby interacting with the device in much the same way that end-users or manual testers might do. The test executor first translates the coded test scripts into machine-executable commands for the robot, and executes them on a robotic arm.

The arm interacts with the mobile device non-intrusively, just as a human would. This process requires inverse kinematics and calibration components in order to make the manipulator act accurately. A camera is used to monitor the mobile device states. Image data from the camera is further processed via computer vision techniques, which perform object detection and test oracle comparison.

The overall process data logged in the execution process is finally sent to a test filter To determine whether the candidate test case should be filtered out.

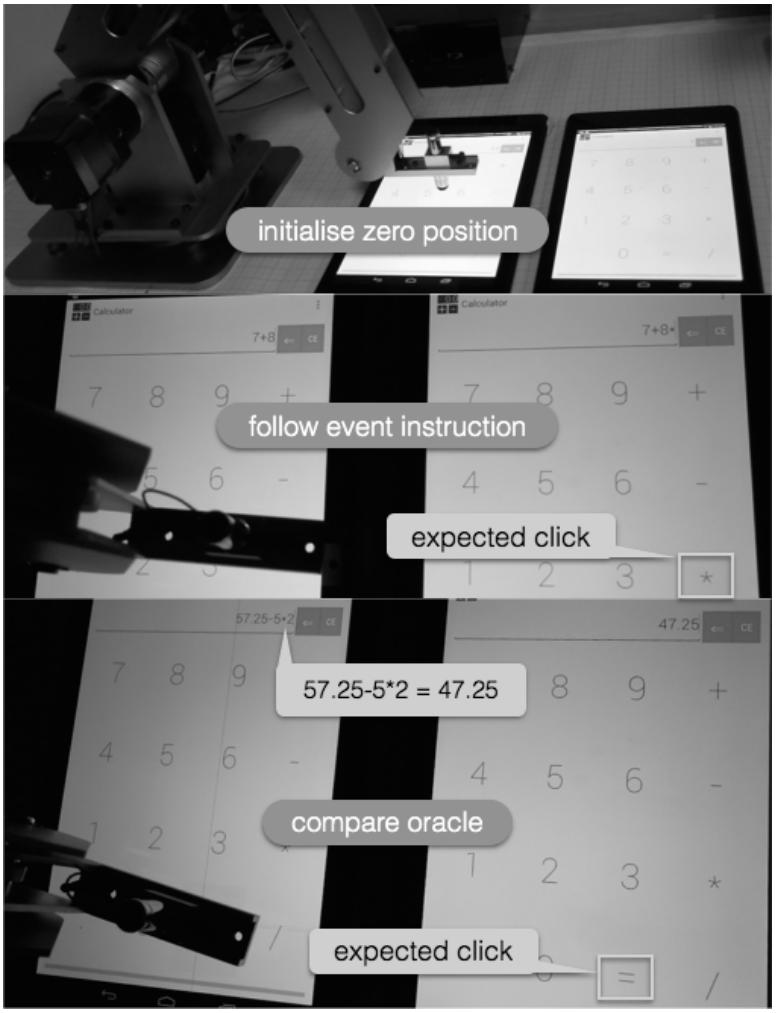



Figure 6.3: Testing a real-world popular calculator app with Axiz[9]

### A Prototype Axiz Implementation

We have implemented a prototype of our Axiz system, shown in Figure 6.2, to demonstrate practical current feasibility. Our implementation has been built entirely using commodity hardware components, which are inexpensive, widely available and interchangeable. We use a 3D vision-based self-calibration approach [274] to help calibrate and adjust the robotic manipulator, in order to keep the system working reliably.

More specifically, we use a 4-axis Arduino-based robotic arm as the manipulator. The arm is driven by stepper motors with a position repeatability of 0.2mm. The maximum speed-of-movement for each axis ranges from 115 to 210 degrees per second (when loaded with a 200g load, a sufficient maximum for most current mobile devices). At the end of robotic forearm, a 'stylus pen' is installed to simulate a human's finger-based

gestures.

We use a Nexus 7 tablet as the device under test, with normal user permissions and the official Android system (without modification) as the platform and operating system. An external CMOS 1,080p camera is used to monitor the test execution. We run the test generator and robot controller on a MacBook Pro laptop with a 2.3 GHz CPU and 16G RAM.

We implemented inverse kinematics (in Python) for robotic arm control. We implemented the object detector and oracle comparator on top of the on OpenCV library. The test generation component implements a widely used multi-objective genetic algorithm called NSGA-II for multi-objective search based software testing, using our (currently state-of-the-art [248]) tool Sapienz, for generating sequences of test events that achieve high coverage and fault revelation with minimised test sequence length.

### 6.2.1 A Proof-of-Concept Illustrative Example

We selected the popular Google Calculator app as our sample subject, which has 5 to 10 million installs according to Google Play[2]. Although this is a simple app, it is a non-trivial real-world app and therefore serves to illustrate the potential for truly black box Robotic Testing.

To demonstrate the usefulness of Axiz, we first use Axiz' robotic test generator to generate realistic tests, which we then execute using the Axiz robotic manipulator. For a comparison purposes, in our demonstration, we also introduce another Nexus 7 tablet (for which more traditional intrusive testing is permitted). This comparator Nexus 7 is directly connected to the PC controller. We allow the test tool for it to have developer-level privileges and permit it to perform OS modifications.

An illustration of this process is shown in Figure 6.3: The interpreter component on the PC translates the event instructions into motion specifications for Axiz' robotic arm controller, which transforms these into joint angle instructions, based on inverse kinematics. As shown in the screenshot, the robotic arm touches the buttons shown

[2]https://play.google.com/store/apps/details?id=com.google.android.calculator

in the left-hand device for robotic test execution. The oracle comparison component 'witnesses' each test event; after each step of the test execution, it captures images via the external camera and validates mobile GUI states. A demo video of Axiz is available online[3], which shows that Axiz was able to accurately execute each test event specified in the generated robotic test cases and to pass all required oracle checkpoints, faithfully increasing the abilities of our traditional machine-to-machine 'blackbox' tester, Sapienz (as described in Chapter 3).

[3] `https://www.youtube.com/watch?v=5SjDAQGloXc`

# Chapter 7

# Conclusions

Mobile testing is of critical importance for app developers to reveal issues before they release the app, yet it is recognised as an expensive and time-consuming process. The state of practice of mobile testing still heavily relies on manual testing. Automating the mobile testing process not only saves human effort, but also increases scalability, which is important because each test may need to be executed on a large number of mobile device models. In addition to the inherent complexity of generating mobile interactions that can navigate apps, mobile test generation is also challenging due to several competing properties that the developers care about, such as test sequence length, code coverage and fault detection capability.

Multi-objective optimisation methods have been extensively applied for solving engineering problems. However, no previous work has proposed a framework to support multi-objective automated testing of mobile apps, nor has previous work yet investigated whether multi-objective search could form a practical approach for testing real-world mobile apps. This thesis introduces the first Pareto multi-objective approach to automated mobile testing, combining techniques used for traditional automated testing, adapting and extending them for Android and JavaScript app testing.

The principle contribution of the thesis is the multi-objective search-based approach for automated mobile testing, i.e., Sapienz for testing Android apps and Octopuz for JavaScript testing. We have demonstrated the superiority of these two automated

test generation techniques over current state of the art and state of practice for both Android and JavaScript app testing. We have also shown its usefulness by using it to test the top 1,000 most popular Google Play apps, where 558 previous unknown faults were revealed using our approach.

Another contribution of the thesis is our investigation of crowdsourcing to assist remote mobile testing and to improve prior test automation approaches by learning from crowd intelligence. We have proposed the POLARIZ approach, which evaluated in the first empirical study on crowdsourced mobile testing via Amazon Mechanical Turk's non-professional crowd. We have also demonstrated its usefulness in harnessing crowd intelligence to enhance our SAPIENZ approach with its automatically learned motif events from the crowdsourced test event traces.

Finally, a subsidiary contribution of the thesis is the first comprehensive survey and analysis of the existing work on crowdsourcing for software engineering. This literature is widely dispersed through many unrelated journal and conference venues. Our survey draws this work together, identifying trends and issues, and collecting existing scientific findings.

The multi-objective search-based mobile testing techniques presented in this thesis have been evaluated by testing apps with real-world complexity. The empirical evaluation results suggest that our proposed multi-objective automated mobile testing approach can help developers to reveal app crashes with minimised test sequence and maximised coverage that it can find. With the assistance of computational search intelligence and crowdsourced human intelligence, we believe search-based mobile test automation will greatly contribute to saving app developers' time in the labour-intensive manual testing activities. It is our hope that this thesis will stimulate future research to optimise additional mobile testing objectives such as realism, and also to consider crowdsourcing as a way of boosting the performance of SBSE techniques.

# Bibliography


[1] "ELLA: A tool for binary instrumentation of Android apps," http://github.com/saswatanand/ella.

[2] "EMMA: A free Java code coverage tool," http://emma.sourceforge.net.

[3] "Android dashboards," http://developer.android.com/about/dashboards/index.html.

[4] "Android fragmentation visualized," http://opensignal.com/reports/2015/08/android-fragmentation.

[5] "Appium: Automation for iOS and Android apps," http://appium.io.

[6] "Dynodroid user guide," http://code.google.com/p/dyno-droid.

[7] "F-Droid," http://f-droid.org.

[8] "Number of Android applications," http://www.appbrain.com/stats/number-of-android-apps.

[9] "Modisco," http://www.eclipse.org/modisco.

[10] "Openstf," https://github.com/openstf/stf.

[11] "Robotium: User scenario testing for Android," https://github.com/RobotiumTech/robotium.

[12] A. Abran, J. W. Moore *et al.*, "Guide to the software engineering body of knowledge." 2004 Version, IEEE CS Professional Practices Committee, 2004.

[13] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of Android test suites in adverse conditions," in *Proc. of ISSTA'15*, 2015, pp. 83–93.

[14] A. Adepetu, K. Ahmed, and Y. A. Abd, "CrowdREquire: A requirements engineering crowdsourcing platform," AAAI, Tech. Rep. SS-12-06, 2012.

[15] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," *Proc. of ISSTA'13*, pp. 352–361, 2013.

[16] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[17] Y. Agarwal and M. Hall, "ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing," in *Proc. of MobiSys'13*, 2013, pp. 97–110.

[18] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *Proc. of USENIX Security'13*, 2013, pp. 257–272.

[19] P. Akiki, A. Bandara, and Y. Yu, "Crowdsourcing user interface adaptations for minimizing the bloat in enterprise applications," in *Proc. of EICS'13*, 2013, pp. 121–126.

[20] R. Ali, C. Solis, M. Salehie, I. Omoronyia, B. Nuseibeh, and W. Maalej, "Social sensing: When users become monitors," in *Proc. of FSE'11*, 2011, pp. 476–479.

[21] R. Ali, C. Solis, I. Omoronyia, M. Salehie, and B. Nuseibeh, "Social adaptation: When software gives users a voice," in *Proc. of ENASE'12*, June 2012, pp. 75–84.

[22] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering*, pp. 742–762, 2010.

[23] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, H. Motahari-Nezhad, E. Bertino, and S. Dustdar, "Quality control in crowdsourcing systems: Issues and directions," *IEEE Internet Computing*, vol. 17, no. 2, pp. 76–81, March 2013.

[24] M. Almaliki, C. Ncube, and R. Ali, "The design of adaptive acquisition of users feedback: An empirical study," in *Proc. of RCIS'14*, 2014, pp. 1–12.

[25] O. Alonso, D. E. Rose, and B. Stewart, "Crowdsourcing for relevance evaluation," in *ACM SigIR Forum*, vol. 42, no. 2, 2008, pp. 9–15.

[26] N. Alshahwan and M. Harman, "Automated Web application testing using search based software engineering," in *Proc. of ASE'11*, 2011, pp. 3–12.

[27] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[28] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon,

"Using GUI ripping for automated testing of Android applications," in *Proc. of ASE'12*, 2012, pp. 258–261.

[29] S. Amann, S. Proksch, and M. Mezini, "Method-call recommendations from implicit developer feedback," in *Proc. of CSISE'14*, June 2014, pp. 5–6.

[30] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. of ESEC/FSE'12*, 2012, pp. 59:1–59:11.

[31] J. H. Andrews, L. C. Briand, and Y. Labiche, in *Proc. of ICSE'05*, May.

[32] M. Aparicio, C. J. Costa, and A. S. Braga, "Proposing a system to support crowdsourcing," in *Proc. of OSDOC'12*, 2012, pp. 13–17.

[33] N. Archak, "Money, glory and cheap talk: Analyzing strategic behavior of contestants in simultaneous crowdsourcing contests on TopCoder.com," in *Proc. of WWW'10*, 2010, pp. 21–30.

[34] A. Arcuri, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 497–519, May 2012.

[35] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. of ICSE'11*, 2011, pp. 1–10.

[36] C. Arellano, O. Díaz, and J. Iturrioz, "Crowdsourced web augmentation : A security model," in *Proc. of ICISE'10*, 2010, pp. 294–307.

[37] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proc. of ISSTA'08*, 2008, pp. 261–272.

[38] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proc. of ICSE'11*, 2011, pp. 571–580.

[39] F. Asadi, G. Antoniol, and Y. Guéhéneuc, "Concept location with genetic algorithms: A comparison of four distributed architectures," in *Proc. of SSBSE'10*, 2010, pp. 153–162.

[40] R. Auler, E. Borin, and P. D. Halleux, "Addressing JavaScript JIT engines performance quirks : A crowdsourced adaptive compiler," in *Proc. of CC'14*, 2014, pp. 218–237.

[41] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. of OOPSLA'13*, 2013, pp. 641–660.

[42] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic search-based testing," in *Proc. of ASE'11)*, 2011, pp. 53–62.

[43] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *Proc. of RSSE'12*, June 2012, pp. 26–30.

[44] J. Bach, "Exploratory testing," in *The Testing Practitioner*, 2004, pp. 253–265.

[45] D. F. Bacon, Y. Chen, D. Parkes, and M. Rao, "A market-based approach to software evolution," in *Proc. of OOPSLA'09*, 2009, pp. 973–980.

[46] S. Bahl and M. Chaturvedi, "Literature review of mobile applications testing on cloud from information security perspective," *International Journal of Computer Applications*, vol. 79, no. 14, 2013.

[47] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "Wsdl-based automatic test case generation for web services testing," in *Proc. of SOSE'05*, 2005, pp. 207–212.

[48] T. Ball, S. Burckhardt, J. de Halleux, M. Moskal, and N. Tillmann, "Beyond open source: The touchdevelop cloud-based integrated development and runtime environment," Tech. Rep. MSR-TR-2014-63, May 2014.

[49] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[50] O. Barzilay, C. Treude, and A. Zagalsky, "Facilitating crowd sourced software engineering via stack overflow," in *Finding Source Code on the Web for Remix and Reuse*. Springer New York, 2013, pp. 289–308.

[51] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, "Motivation in software engineering: A systematic literature review," *Information and Software Technology*, vol. 50, no. 9, pp. 860–878, 2008.

[52] A. Begel, R. DeLine, and T. Zimmermann, "Social media for software engineering," in *Proc. of FoSER'10*, 2010, pp. 33–38.

[53] A. Begel, J. Bosch, and M.-A. Storey, "Social networking meets software development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder," *IEEE Software*, vol. 30, no. 1, pp. 52–66, January 2013.

[54] B. Bergvall-Kå reborn and D. Howcroft, "The Apple business model: Crowd-

sourcing mobile applications," *Accounting Forum*, vol. 37, no. 4, pp. 280–289, December 2013.

[55] M. S. Bernstein, "Crowd-powered interfaces," in *Proc. of UIST'10*, 2010, pp. 347–350.

[56] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, "Code Hunt: Experience with coding contests at scale," *Proc. of ICSE'15 (JSEET)*, June 2015.

[57] R. Blanco, H. Halpin, D. M. Herzig, P. Mika, J. Pound, H. S. Thompson, and T. Tran Duc, "Repeatable and reliable search system evaluation using crowdsourcing," in *Proc. of SIGIR'11*, 2011, pp. 923–932.

[58] B. W. Boehm *et al.*, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[59] M. N. K. Boulos, B. Resch, D. N. Crowley, J. G. Breslin, G. Sohn, R. Burtner, W. A. Pike, E. Jezierski, and K.-Y. S. Chuang, "Crowdsourcing, citizen sensing and sensor web technologies for public and environmental health surveillance and crisis management: trends, OGC standards and application examples," *International journal of health geographics*, vol. 10, no. 1, p. 67, 2011.

[60] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *Proc. of SOSE'11*, 2011, pp. 13–24.

[61] D. C. Brabham, "Crowdsourcing as a model for problem solving an introduction and cases," *Convergence: the international journal of research into new media technologies*, vol. 14, no. 1, pp. 75–90, 2008.

[62] D. C. Brabham, T. W. Sanchez, and K. Bartholomew, "Crowdsourcing public participation in transit planning: preliminary results from the next stop design case," *Transportation Research Board*, 2009.

[63] T. D. Breaux and F. Schaub, "Scaling requirements extraction to the crowd: Experiments with privacy policies," in *Proc. of RE'14*, August 2014, pp. 163–172.

[64] N. Breslow, "A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship," *Biometrika*, vol. 57, no. 3, pp. 579–594, 1970.

[65] L. C. Briand, J. Feng, and Y. Labiche, "Using genetic algorithms and coupling measures to devise optimal integration test orders," in *Proc. of SEKE'02*, 2002, pp. 43–50.

[66] B. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. of GECCO'15*, Madrid, Spain, July 2015, pp. 1327–1334.

[67] M. Bruch, "IDE 2.0: Leveraging the wisdom of the software engineering crowds," Ph.D. dissertation, Technische Universität Darmstadt, 2012.

[68] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini, "IDE 2.0: Collective intelligence in software development," in *Proc. of FoSER'10*, ser. FoSER '10, 2010, pp. 53–58.

[69] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. of SPSM'11*, 2011, pp. 15–26.

[70] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, February 2013.

[71] E. Cantú-Paz and D. E. Goldberg, "Efficient parallel genetic algorithms: theory and practice," *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2–4, pp. 221–238, 2000.

[72] S. Carino and J. H. Andrews, "Dynamically testing GUIs using ant colony optimization," in *Proc. of ASE'15*, 2015, pp. 138–148.

[73] C. Challiol, S. Firmenich, G. A. Bosetti, S. E. Gordillo, and G. Rossi, "Crowdsourcing mobile web applications," in *Proc. of ICWE'13 Workshops*, 2013, pp. 223–237.

[74] A. T. Chatfield and U. Brajawidagda, "Crowdsourcing hazardous weather reports from citizens via twittersphere under the short warning lead times of EF5 intensity tornado conditions," in *Proc. of HICSS'14*, 2014, pp. 2231–2241.

[75] C. Chen and K. Zhang, "Who asked what: Integrating crowdsourced FAQs into API documentation," in *Proc. of ICSE'14 (Companion)*, 2014, pp. 456–459.

[76] F. Chen and S. Kim, "Crowd debugging," in *Proc. of FSE'15*, 2015, pp. 320–332.

[77] K.-t. Chen, C.-j. Chang, A. Sinica, C.-c. Wu, Y.-c. Chang, and C.-l. Lei, "Quadrant of Euphoria: A crowdsourcing platform for QoE assessment," *IEEE Network*, no. April, pp. 28–35, 2010.

[78] N. Chen and S. Kim, "Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles," in *Proc. of ASE'12*, 2012, pp. 140–149.

[79] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "AR-Miner: Mining informative reviews for developers from mobile app marketplace," in *Proc. of ICSE'14*,

2014, pp. 767–778.

[80] P. K. Chilana, "Supporting users after software deployment through selection-based crowdsourced contextual help," Ph.D. dissertation, University of Washington, 2013.

[81] P. K. Chilana, A. J. Ko, and J. O. Wobbrock, "LemonAid: Selection-based crowdsourced contextual help for web applications," in *Proc. of CHI'12*, 2012, pp. 1549–1558.

[82] P. K. Chilana, A. J. Ko, J. O. Wobbrock, and T. Grossman, "A multi-site field study of crowdsourced contextual help : Usage and perspectives of end users and software teams," in *Proc. of CHI'13*, 2013.

[83] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. of OOPSLA'13*, 2013, pp. 623–640.

[84] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proc. of ASE'15*, 2015, pp. 429–440.

[85] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing gui event traces," in *Proc. of FSE'16*, 2016, pp. 422–434.

[86] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, M. Gethers, D. Poshyvanyk, J. H. Hayes, and W. Li, "Toward actionable, broadly accessible contests in software engineering," in *Proc. of ICSE'12*, June 2012, pp. 1329–1332.

[87] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," in *Proc. of POPL'15*, 2015, pp. 677–688.

[88] comScore, "Global digital future in focus," http://www.comscore.com/Insights/Presentations-and-Whitepapers/2016/2016-Global-Digital-Future-in-Focus, 2016.

[89] comScore, "The global mobile report," http://comscore.com/Insights/Presentations-and-Whitepapers/2015/The-Global-Mobile-Report, 2015.

[90] C. Cook and M. Visconti, "Documentation is important," *CrossTalk*, vol. 7, no. 11, pp. 26–30, 1994.

[91] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay,

D. Baker, Z. Popović *et al.*, "Predicting protein structures with a multiplayer online game," *Nature*, vol. 466, no. 7307, pp. 756–760, 2010.

[92] H. Dan, M. Harman, J. Krinke, A. Marginean, L. Li, and F. Wu, "Pidgin Crasher: Searching for minimised crashing gui event sequences," in *Proc. of SSBSE'14*, August 2014, pp. 253–258.

[93] M. K. Das and H.-K. Dai, "A survey of dna motif finding algorithms," *BMC bioinformatics*, vol. 8, no. 7, p. 1, 2007.

[94] L. B. L. de Souza, E. C. Campos, and M. D. A. Maia, "Ranking crowd knowledge to assist software development," in *Proc. of ICPC'14*, 2014, pp. 72–82.

[95] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[96] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[97] R. A. DeMillo and A. J. Offutt, "Constraint–based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[98] P. D'haeseleer, "What are dna sequence motifs?" *Nature biotechnology*, vol. 24, no. 4, pp. 423–425, 2006.

[99] J. Dibbern, T. Goles, R. Hirschheim, and B. Jayatilaka, "Information systems outsourcing: a survey and analysis of the literature," *The Data Base for Advances in Information Systems*, vol. 35, no. 4, pp. 6–102, 2004.

[100] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović, "Verification games: Making verification fun," in *Proc. of FTfJP'12*, 2012, pp. 42–49.

[101] E. Dolstra, R. Vliegendhart, and J. Pouwelse, "Crowdsourcing GUI tests," in *Proc. of ISSTA'13*, March 2013, pp. 332–341.

[102] W. Ebner, M. Leimeister, U. Bretschneider, and H. Krcmar, "Leveraging the wisdom of crowds: Designing an IT-supported ideas competition for an ERP software company," in *Proc. of HICSS'08*, January 2008, pp. 417–417.

[103] M. Ermuth and M. Pradel, "Monkey see, monkey do: Effective generation of gui

tests with inferred macro events," in *Proc. of ISSTA'16*, 2016, pp. 82–93.

[104] M. D. Ernst and Z. Popović, "Crowd-sourced program verification," University OF Washington, Tech. Rep., 2012.

[105] B. Esselink, *A practical guide to localization.* John Benjamins Publishing, 2000, vol. 4.

[106] E. Estellés-Arolas and F. González-Ladrón-De-Guevara, "Towards an integrated crowdsourcing definition," *Journal of Information Science*, vol. 38, no. 2, pp. 189–200, April 2012.

[107] C. Exton, A. Wasala, J. Buckley, and R. Schäler, "Micro crowdsourcing: A new model for software localisation," *Localisation Focus*, vol. 8, no. 1, pp. 81–89, 2009.

[108] J. Farrell and M. Rabin, "Cheap talk," *The Journal of Economic Perspectives*, vol. 10, no. 3, pp. 103–118, 1996.

[109] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein, "Emergent, crowd-scale programming practice in the IDE," in *Proc. of CHI'14*, 2014, pp. 2491–2500.

[110] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: multi-objective overtime planning for software engineering projects," in *Proc. of ICSE'13*, 2013, pp. 462–471.

[111] B. Fitzgerald and K.-J. Stol, "The dos and don'ts of crowdsourcing software development," in *SOFSEM 2015: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, 2015, vol. 8939, pp. 58–64.

[112] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, July 2012.

[113] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 8:1–8:42, 2014.

[114] G. Fraser and A. Arcuri, "Handling test length bloat," *Software Testing, Verification and Reliability*, vol. 23, no. 7, pp. 553–582, November 2013.

[115] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[116] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proc.*

*of QSIC'11*, 2011, pp. 31–40.

[117] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proc. of ESEC/FSE'11*, 2011, pp. 416–419.

[118] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *Proc. of ICTS'12*, 2012, pp. 121–130.

[119] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. of ISSTA'10*, 2010, pp. 147–158.

[120] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. of ISSTA'10*. Trento, Italy: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[121] D. Fried, "Crowdsourcing in the software development industry," *Nexus of Entrepreneurship and Technology Initiative*, 2010.

[122] M. T. Frohlich, "Techniques for improving response rates in OM survey research," *Journal of Operations Management*, vol. 20, no. 1, pp. 53–62, 2002.

[123] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," in *Proc. of ICSM'10*, 2010.

[124] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proc. of ISSTA'12*, 2012, pp. 177–187.

[125] B. Gardlo, S. Egger, M. Seufert, and R. Schatz, "Crowdsourcing 2.0: Enhancing execution speed and reliability of web-based QoE testing," in *Proc. of ICC'14*, June 2014, pp. 1070–1075.

[126] Garter, "Gartner says by 2016, more than 50 percent of mobile apps deployed will be hybrid," http://www.gartner.com/newsroom/id/2324917, 2013.

[127] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proc. of ISSTA'16*, 2016, pp. 213–224.

[128] M. Goldman, "Role-based interfaces for collaborative software development," in *Proc. of UIST'11*, 2011, pp. 23–26.

[129] M. Goldman, "Software development with real-time collaborative editing," Ph.D. dissertation, Massachusetts Institute of Technology, 2012.

[130] M. Goldman, G. Little, and R. C. Miller, "Real-time collaborative coding in a web IDE," in *Proc. of UIST'11*, 2011, pp. 155–164.

[131] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing-and touch-sensitive record and replay for android," in *Proc. of ICSE'13*, 2013, pp. 72–81.

[132] V. H. M. Gomide, P. A. Valle, J. O. Ferreira, J. R. G. Barbosa, A. F. da Rocha, and T. M. G. d. A. Barbosa, "Affective crowdsourcing applied to usability testing," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 1, pp. 575–579, 2014.

[133] Google, "Android Monkey," http://developer.android.com/tools/help/monkey.html.

[134] P. Greenwood, A. Rashid, and J. Walkerdine, "UDesignIt: Towards social media for community-driven design," *Proc. of ICSE'12*, pp. 1321–1324, June 2012.

[135] A. Gritti, "Crowd outsourcing for software localization," Master's thesis, Universitat Politécnica de Catalunya, 2012.

[136] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *Proc. of ISSTA'12*, 2012, pp. 67–77.

[137] S. Gueorguiev, M. Harman, and G. Antoniol, "Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering," in *Proc. of GECCO'09*, 2009, pp. 1673–1680.

[138] E. Guzman and W. Maalej, "How do users like this feature? A fine grained sentiment analysis of app reviews," in *Proc. of RE'14*, Aug 2014, pp. 153–162.

[139] W. G. J. Halfond and A. Orso, "Improving test case generation for web applications using automated interface discovery," in *Proc. of FSE'07*, 2007, pp. 145–154.

[140] S. Hamidi, P. Andritsos, and S. Liaskos, "Constructing adaptive configuration dialogs using crowd data," in *Proc. of ASE'14*, 2014, pp. 485–490.

[141] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

[142] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. of MobiSys'14*, 2014, pp. 204–217.

[143] M. Harman, "The current state and future of search based software engineering," in *Proc. of FOSE'07*, 2007, pp. 342–357.

[144] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," in *Proc. of Regression'11*, Berlin, Germany, March 2011.

[145] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[146] M. Harman and B. F. Jones, *Information and Software Technology*, December.

[147] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: MSR for App Stores," in *Proc. of MSR'12*, Zurich, Switzerland, 2-3 June 2012.

[148] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–11:61, November 2012.

[149] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification: LASER 2009-2010*, B. Meyer and M. Nordio, Eds., 2012, pp. 1–59, LNCS 7007.

[150] M. Harman, W. B. Langdon, and Y. Jia, "Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system," in *Proc. of SSBSE'14*. Fortaleza, Brazil: Springer LNCS, August 2014, pp. 247–252.

[151] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Proc. of ICST'15*, Graz, Austria, April 2015.

[152] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Proc. of ICST'15*, 2015, pp. 1–12.

[153] R. Harrison, D. Flood, and D. Duce, "Usability of mobile applications: literature review and rationale for a new usability model," *Journal of Interaction Science*, vol. 1, no. 1, p. 1, 2013.

[154] B. Hartmann, D. Macdougall, J. Brandt, and S. R. Klemmer, "What would other programmers do? Suggesting solutions to error messages," in *Proc. of CHI'10*, 2010, pp. 1019–1028.

[155] H. He, Z. Ma, H. Chen, and W. Shao, "How the crowd impacts commercial applications: A user-oriented approach," in *Proc. of CrowdSoft'14*, 2014, pp. 1–6.

[156] A. T. Holdener, III, *Ajax: The Definitive Guide*, 1st ed. USA: O'Reilly Media, Inc., 2008.

[157] M. Hosseini, K. Phalp, J. Taylor, and R. Ali, "Towards crowdsourcing for requirements engineering," in *Proc. of REFSQ'13 (Empirical Track)*, 2013.

[158] M. Hosseini, A. Shahri, K. Phalp, J. Taylor, R. Ali, and F. Dalpiaz, "Configuring crowdsourcing for requirements elicitation," in *Proc. of RCIS'15*, 2015.

[159] T. Hossfeld, C. Keimel, M. Hirth, B. Gardlo, J. Habigt, and K. Diepold, "Best

practices for QoE crowdtesting : QoE assessment with crowdsourcing," *IEEE Transactions on Multimedia*, vol. 16, no. 2, pp. 541–558, 2014.

[160] T. Hossfeld, C. Keimel, and C. Timmerer, "Crowdsourcing quality-of-experience assessments," *Computer*, pp. 98–102, 2014.

[161] J. Howe, "The rise of crowdsourcing," *Wired magazine*, vol. 14, no. 6, pp. 1–4, 2006.

[162] J. Howe, "Crowdsourcing: A definition," http://crowdsourcing.typepad.com/cs/2006/06/crowdsourcing_a.html, June 2006.

[163] Z. Hu and W. Wu, "A game theoretic model of software crowdsourcing," in *Proc. of SOSE'14*, April 2014, pp. 446–453.

[164] Y.-C. Huang, C.-I. Wang, and J. Hsu, "Leveraging the crowd for creating wireframe-based exploration of mobile design pattern gallery," in *Proc. of IUI Companion'13*, 2013, pp. 17–20.

[165] J. M. Hughes, "Systems and methods for software development," August 2010, US Patent 7778866 B2.

[166] M. N. Huhns, W. Li, and W.-T. Tsai, "Cloud-based software crowdsourcing (Dagstuhl seminar 13362)," *Dagstuhl Reports*, vol. 3, no. 9, pp. 34–58, 2013.

[167] H. Huo, Z. Zhao, V. Stojkovic, and L. Liu, "Optimizing genetic algorithm for motif discovery," *Mathematical and Computer Modelling*, vol. 52, no. 11, pp. 2011–2020, 2010.

[168] P. G. Ipeirotis, F. Provost, and J. Wang, "Quality management on amazon mechanical turk," in *Proc. of HCOMP'10*, 2010, pp. 64–67.

[169] Q. Ismail, T. Ahmed, A. Kapadia, and M. K. Reiter, "Crowdsourced exploration of security configurations," in *Proc. of CHI'15*, 2015, pp. 467–476.

[170] J. Itkonen and K. Rautiainen, "Exploratory testing: A multiple case study," in *Proc. of ESEM'05*, 2005, pp. 84–93.

[171] J. Itkonen, M. V. Mantyla, and C. Lassenius, "How do testers do it? an exploratory study on manual testing practices," in *Proc. of ESEM'09*, 2009, pp. 494–497.

[172] J. Itkonen, M. V. Mantyla, and C. Lassenius, "The role of the tester's knowledge in exploratory software testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 707–724, 2013.

[173] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment

and improvement," in *Proc. of ISSTA'16*, 2016, pp. 247–258.

[174] S. James, "The wisdom of the crowds," *New York: Randome House*, 2004.

[175] R. Jayakanthan and D. Sundararajan, "Enterprise crowdsourcing solution for software development in an outsourcing organization," in *Proc. of ICWE'11*, 2011, pp. 177–180.

[176] R. Jayakanthan and D. Sundararajan, "Enterprise crowdsourcing solutions for software development and ideation," in *Proc. of UbiCrowd'11*, 2011, pp. 25–28.

[177] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proc. of ISSTA'13*, 2013, pp. 67–77.

[178] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[179] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[180] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proc. of ICSE'15*, 2015, pp. 540–550.

[181] H. C. Jiau and F.-P. Yang, "Facing up to the inequality of crowdsourced API documentation," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 1, pp. 1–9, January 2012.

[182] R. Johnson, "Natural products: Crowdsourcing drug discovery," *Nature chemistry*, vol. 6, no. 2, pp. 87–87, 2014.

[183] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Proc. of ESEM'13*, 2013, pp. 15–24.

[184] N. Juristo and S. Vegas, "The role of non-exact replications in software engineering experiments," *Empirical Software Engineering*, vol. 16, no. 3, pp. 295–324, 2011.

[185] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. of FSE'14*, November 2014, pp. 654–665.

[186] M. Kajko-Mattsson, "A survey of documentation practice within corrective main-

tenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[187] M. Kallenbach, "HelpMeOut-Crowdsourcing suggestions to programming problems for dynamic, interpreted languages," Master's thesis, RWTH Aachen University, 2011.

[188] C. Kaner, J. Bach, and B. Pettichord, *Lessons learned in software testing*, 2008.

[189] M. D. Kaplowitz, T. D. Hadlock, and R. Levine, "A comparison of web and mail survey response rates," *Public Opinion Quarterly*, vol. 68, no. 1, pp. 94–101, 2004.

[190] M. Kaya, "Mogamod: Multi-objective genetic algorithm for motif discovery," *Expert Systems with Applications*, vol. 36, no. 2, pp. 1039–1047, 2009.

[191] R. Kazman and H.-M. Chen, "The metropolis model a new logic for development of crowdsourced systems," *Communications of the ACM*, vol. 52, no. 7, pp. 76–84, 2009.

[192] R. Kazman and H.-M. Chen, "The metropolis model and its implications for the engineering of software ecosystems," in *Proc. of FoSER'10*, 2010, pp. 187–190.

[193] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[194] F. Khatib, F. DiMaio, S. Cooper, M. Kazmierczyk, M. Gilski, S. Krzywda, H. Zabranska, I. Pichova, J. Thompson, Z. Popović *et al.*, "Crystal structure of a monomeric retroviral protease solved by protein folding game players," *Nature Structural and Molecular Biology*, vol. 18, no. 10, pp. 1175–1177, 2011.

[195] J. Kim, I. Lee, Y. Lee, B. Choi, S.-J. Hong, K. Tam, K. Naruse, and Y. Maeda, "Exploring e-business implications of the mobile internet: a cross-national survey in hong kong, japan and korea," *International Journal of Mobile Communications*, vol. 2, no. 1, pp. 1–21, 2004.

[196] B. Kitchenham, T. Dybå, and M. Jørgensen, "Evidence-based software engineering," in *Proc. of ICSE'14*, 2004, pp. 273–281.

[197] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut, "CrowdForge: Crowdsourcing complex work," in *Proc. of UIST'11*, 2011, pp. 43–52.

[198] P. S. Kochhar, F. Thung, N. Nagappan, and T. Zimmermann, "Understanding the test automation culture of app developers," in *Proc. of ICST'15*, 2015, pp. 1–10.

[199] B. Kogut and A. Metiu, "Open-source software development and distributed

innovation," *Oxford Review of Economic Policy*, vol. 17, no. 2, pp. 248–264, 2001.

[200] H. Krcmar, U. Bretschneider, M. Huber, and J. M. Leimeister, "Leveraging crowdsourcing: Activation-supporting components for IT-based ideas competition," *Journal of Management Information Systems*, vol. 26, no. 1, pp. 197–224, 2009.

[201] K. R. Lakhani, D. A. Garvin, and E. Lonstein, "TopCoder(A): Developing software through crowdsourcing," *Harvard Business School Case*, 610-032, January 2010.

[202] K. R. Lakhani, K. J. Boudreau, P.-R. Loh, L. Backstrom, C. Baldwin, E. Lonstein, M. Lydon, A. MacCormack, R. A. Arnaout, and E. C. Guinan, "Prize-based contests can provide solutions to computational biology problems," *Nature Biotechnology*, vol. 31, no. 2, pp. 108–111, 2013.

[203] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: Haven't we solved this problem yet?" in *Proc. of 4th Testing Academia and Industry Conference — Practice And Research Techniques*, 4th–6th September 2009, pp. 95–104.

[204] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[205] W. B. Langdon, B. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Proc. of GECCO'15*, Madrid, Spain, July 2015, pp. 1063–1070.

[206] W. S. Lasecki, J. Kim, N. Rafter, O. Sen, J. P. Bigham, and M. S. Bernstein, "Apparition: Crowdsourced user interfaces that come to life as you sketch them," in *Proc. of CHI'15*, 2015, pp. 1925–1934.

[207] T. D. LaToza and A. van der Hoek, "A vision of crowd development," in *Proc. of ICSE'15 (NIER Track)*, 2015.

[208] T. D. LaToza, W. Ben Towne, A. van der Hoek, and J. D. Herbsleb, "Crowd development," in *Proc. of CHASE'13*, May 2013, pp. 85–88.

[209] T. D. LaToza, E. Chiquillo, W. Ben Towne, C. Adriano, and A. van der Hoek, "CrowdCode - crowd development," in *Proc. of CrowdConf'13*, 2013.

[210] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek, "Microtask programming: Building software with a crowd," in *Proc. of UIST'14*, 2014, pp. 43–54.

[211] T. D. LaToza, W. B. Towne, and A. V. D. Hoek, "Harnessing the crowd : Decontextualizing software work," in *Proc. of CSD'14*, 2014, pp. 2–3.

[212] T. D. LaToza, M. Chen, L. Jiang, M. Zhao, and A. V. D. Hoek, "Borrowing from the crowd : A study of recombination in software design competitions," in *Proc. of ICSE'15*, 2015.

[213] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[214] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[215] M. Lease and E. Yilmaz, "Crowdsourcing for information retrieval," in *ACM SIGIR Forum*, vol. 45, no. 2, 2012, pp. 66–75.

[216] Y. Lee and K. A. Kozar, "Investigating the effect of website quality on e-business success: An analytic hierarchy process (AHP) approach," *Decision support systems*, vol. 42, no. 3, pp. 1383–1401, 2006.

[217] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proc. of ASE'07*, 2007, pp. 417–420.

[218] S. Leone, "Information components as a basis for crowdsourced information system development," Ph.D. dissertation, Swiss Federal Institute of Technology in Zurich, 2011.

[219] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Proc. of ICSE'14*. New York, NY, USA: ACM, 2014, pp. 527–538.

[220] K. Li, J. Xiao, Y. Wang, and Q. Wang, "Analysis of the key factors for software quality in crowdsourcing development: An empirical study on TopCoder.com," in *Proc. of COMPSAC'13*, 2013, pp. 812–817.

[221] W. Li, S. Seshia, and S. Jha, "CrowdMine: Towards crowdsourced human-assisted verification," in *Proc. of DAC'12*, 2012, pp. 2–3.

[222] W. Li, M. N. Huhns, W.-T. Tsai, and W. Wu, *Crowdsourcing: Cloud-Based*

*Software Development*. Springer, 2015.

[223] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa : Automated large-scale mobile app testing through contextual fuzzing," in *Proc. of MobiCom'14*, 2014.

[224] S. Lim, D. Quercia, and A. Finkelstein, "StakeNet: Using social networks to analyse the stakeholders of large-scale software projects," *Proc. of ICSE'10*, vol. 2010, 2010.

[225] S. L. Lim, "Social networks and collaborative filtering for large-scale requirements elicitation," Ph.D. dissertation, University of New South Wales, 2010.

[226] S. L. Lim and A. Finkelstein, "StakeRare: Using social networks and collaborative filtering for large-scale requirements elicitation," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 707–735, 2012.

[227] S. L. Lim and C. Ncube, "Social networks and crowdsourcing for stakeholder analysis in system of systems projects," in *Proc. of SOSE'13*, June 2013, pp. 13–18.

[228] S. L. Lim, D. Quercia, and A. Finkelstein, "StakeSource: Harnessing the power of crowdsourcing and social networks in stakeholder analysis," in *Proc. of ICSE'10*, vol. 2, 2010, pp. 239–242.

[229] S. L. Lim, D. Damian, and A. Finkelstein, "StakeSource2.0: Using social networks of stakeholders to identify and prioritise requirements," in *Proc. of ICSE'11*, 2011, pp. 1022–1024.

[230] J. Lin, "Understanding and capturing people's mobile app privacy preferences," Ph.D. dissertation, Carnegie Mellon University, 2013.

[231] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *Proc. of Ubicomp'12*, 2012, pp. 501–510.

[232] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for Android devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, October 2014.

[233] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execu-

tion scenarios," in *Proc. of MSR'15*, 2015, pp. 111–122.

[234] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execution scenarios," in *Proc. of MSR'15*, 2015, pp. 111–122.

[235] D. Liu, R. G. Bias, M. Lease, and R. Kuipers, "Crowdsourcing for usability testing," in *Proc. of ASIST'12*, vol. 49, no. 1, January 2012, pp. 1–10.

[236] M. Lydon, "Topcoder overview," http://www.nasa.gov/pdf/651447main_TopCoder_Mike_D1_830am.pdf, 2012, Accessed: 2015-11-23.

[237] W. Maalej and D. Pagano, "On the socialness of software," in *Proc. of DASC'11*, Dec 2011, pp. 864–871.

[238] L. Machado, G. Pereira, R. Prikladnicki, E. Carmel, and C. R. B. de Souza, "Crowdsourcing in the Brazilian it industry: What we know and what we don't know," in *Proc. of CrowdSoft'14*, 2014, pp. 7–12.

[239] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. of ESEC/FSE'13*, 2013, pp. 224–234.

[240] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented evolutionary testing of Android apps," in *Proc. of FSE'14*, 2014, pp. 599–609.

[241] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. of ICSE'14*. New York, NY, USA: ACM, 2014, pp. 503–514.

[242] N. Mansour and M. Houri, "Testing web applications," *Information and Software Technology*, vol. 48, no. 1, pp. 31–42, 2006.

[243] M. V. Mäntylä and J. Itkonen, "More testers - the effect of crowd size and time restriction in software testing," *Information and Software Technology*, vol. 55, no. 6, pp. 986–1003, June 2013.

[244] J. Manzoor, "A crowdsourcing framework for software localization," Master's thesis, KTH Royal Institute of Technology, 2011.

[245] K. Mao, Y. Yang, M. Li, and M. Harman, "Pricing Crowdsourcing Based Software Development Tasks," in *Proc. of ICSE'13 (NIER Track)*, 2013, pp. 1205–1208.

[246] K. Mao, Y. Yang, Q. Wang, Y. Jia, and M. Harman, "Developer recommendation for crowdsourced software development tasks," in *Proc. of SOSE'15*, 2015, pp. 347–356.

[247] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing

in software engineering," *Journal of Systems and Software*, 2016.

[248] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. of ISSTA'16*, 2016, pp. 94–105.

[249] A. Marchetto and P. Tonella, "Search-based testing of ajax web applications," in *SSBSE'09*, 2009, pp. 3–12.

[250] A. Marchetto and P. Tonella, "Using search-based algorithms for ajax event sequence generation during testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2010.

[251] Marmelab, "Gremlins: Monkey testing library for web apps and node.js," http://github.com/marmelab/gremlins.js, 2013.

[252] E. Martin, S. Basu, and T. Xie, "Websob: A tool for robustness testing of web services," in *Proc. of ICSE'07 Companion*, 2007, pp. 65–66.

[253] S. F. Martin, H. Falkenberg, T. F. Dyrlund, G. A. Khoudoli, C. J. Mageean, and R. Linding, "PROTEINCHALLENGE: crowd sourcing in proteomics analysis and software development." *Journal of Proteomics*, vol. 88, pp. 41–6, August 2013.

[254] W. Martin, "Causal impact analysis for app releases in google play," in *Proc. of FSE'16*, 2016, pp. 435–446.

[255] Massolution, "Crowdsourcing industry report," http://www.crowdsourcing.org/editorial/enterprise-crowdsourcing-trends-infographic/18725, 2012, Accessed: 2015-03-01.

[256] P. McMinn, "Search-based software testing: Past, present and future," in *Proc. of SBST'11*, 2011, pp. 153–163.

[257] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[258] F. Meier, A. Bazo, M. Burghardt, and C. Wolff, "Evaluating a web-based tool for crowdsourced navigation stress tests," in *Proc. of 2nd International Conference on Design, User Experience, and Usability: Web, Mobile, and Product Design*, 2013, pp. 248–256.

[259] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proc. of WCRE'03*, Nov 2003, pp. 260–269.

[260] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing,"

*ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 256–267, 2001.

[261] A. Méndez-Porras, C. Quesada-López, and M. Jenkins, "Automated testing of mobile applications: a systematic map and review," in *XVIII Ibero-American Conference on Software Engineering, Lima-Peru*, 2015, pp. 195–208.

[262] A. Mijnhardt, "Crowdsourcing for enterprise software localization," Master thesis, Utrecht University, 2013.

[263] P. Minder and A. Bernstein, "CrowdLang - First steps towards programmable human computers for general computation," in *Proc. of HCOMP'11*, JAN 2011, pp. 103–108.

[264] P. Minder and A. Bernstein, "CrowdLang: A programming language for the systematic exploration of human computation systems," in *Proc. of SocInfo'12*. Lausanne: Springer, DEC 2012.

[265] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *Proc. of ICST'13*, March 2013, pp. 74–83.

[266] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "JSeft: Automated JavaScript unit test generation," in *Proc. of ICST'15*, 2015, pp. 1–10.

[267] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[268] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of Android applications," in *Proc. of ICSE'16*, 2016, pp. 559–570.

[269] A. Misra, A. Gooze, K. Watkins, M. Asad, and C. A. Le Dantec, "Crowdsourcing and its application to transportation data collection and management," *Transportation Research Record: Journal of the Transportation Research Board*, vol. 2414, no. 1, pp. 1–8, 2014.

[270] B. S. Mitchell, M. Traverso, and S. Mancoridis, "An architecture for distributing the computation of software clustering algorithms," in *Proc. of WICSA'01*, 2001, pp. 181–190.

[271] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 17:1–

17:45, May 2015.

[272] M. Mooty, A. Faulring, J. Stylos, and B. a. Myers, "Calcite: Completing code completion for constructors using crowds," in *Proc. of VL/HCC'10*, September 2010, pp. 15–22.

[273] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing Android application crashes," in *Proc. of ICST'16*, 2016, pp. 33–44.

[274] J. M. S. Motta, G. C. de Carvalho, and R. McMaster, "Robot calibration using a 3D vision-based measurement system with a single camera," *Robotics and Computer-Integrated Manufacturing*, vol. 17, no. 6, pp. 487 – 497, 2001.

[275] N. Muganda, D. Asmelash, and S. Mlay, "Groupthink decision making deficiency in the requirements engineering process: Towards a crowdsourcing model," *SSRN Electronic Journal*, 2012.

[276] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann, "Crowdsourcing suggestions to programming problems for dynamic web development languages," in *Proc. of CHI'11*, 2011, pp. 1525–1530.

[277] C. Muller, L. Chapman, S. Johnston, C. Kidd, S. Illingworth, G. Foody, A. Overeem, and R. Leigh, "Crowdsourcing for climate and atmospheric sciences: current status and future potential," *International Journal of Climatology*, 2015.

[278] R. Musson, J. Richards, D. Fisher, C. Bird, B. Bussone, and S. Ganguly, "Leveraging the Crowd: How 48,000 Users Helped Improve Lync Performance," *IEEE Software*, vol. 30, no. 4, pp. 38–45, July 2013.

[279] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[280] S. Nag, I. Heffan, A. Saenz-Otero, and M. Lydon, "SPHERES Zero Robotics software development: Lessons on crowdsourcing and collaborative competition," in *Proc. of IEEE Aerospace Conference'12*, March 2012, pp. 1–17.

[281] S. Nag, "Collaborative competition for crowdsourcing spaceflight software and STEM education using SPHERES Zero Robotics," Master's thesis, Massachusetts Institute of Technology, 2012.

[282] P. Nascimento, R. Aguas, D. Schneider, and J. de Souza, "An approach to requirements categorization using Kano's model and crowds," in *Proc. of CSCWD'12*,

May 2012, pp. 387–392.

[283] M. Nayebi and G. Ruhe, "An open innovation approach in support of product release decisions," in *Proc. of CHASE'14*, 2014, pp. 64–71.

[284] M. Nebeling and M. C. Norrie, "Context-aware and adaptive web interfaces : A crowdsourcing approach," in *Proc. of ICWE'11*, 2011, pp. 167–170.

[285] M. Nebeling and M. C. Norrie, "Tools and architectural support for crowdsourced adaptation of web interfaces," in *Proc. of ICWE'11*, 2011, pp. 243–257.

[286] M. Nebeling, S. Leone, and M. Norrie, "Crowdsourced web engineering and design," in *Proc. of ICWE'12*, 2012, pp. 1–15.

[287] M. Nebeling, M. Speicher, M. Grossniklaus, and M. C. Norrie, "Crowdsourced web site evaluation with crowdstudy," in *Proc. of ICWE'12*, 2012, pp. 494–497.

[288] M. Nebeling, M. Speicher, and M. C. Norrie, "CrowdStudy: General toolkit for crowdsourced evaluation of web interfaces," in *Proc. of EICS'13*, 2013.

[289] M. Nebeling, M. Speicher, and M. C. Norrie, "CrowdAdapt: Enabling crowdsourced web page adaptation for individual viewing conditions and preferences," in *Proc. of EICS'13*, 2013, pp. 23–32.

[290] G. Neumann, M. Harman, and S. M. Poulding, "Transformed vargha-delaney effect size," in *Proc. of SSBSE'15*, Bergamo, Italy, September 2015, pp. 318–324.

[291] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *Proc. of ICSE'13*, B. H. C. Cheng and K. Pohl, Eds. San Francisco, USA: IEEE, May 18-26 2013, pp. 772–781.

[292] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden, "Mutation analysis for javascript web application testing," in *Proc. of SEKE'13*, 2013, pp. 159–165.

[293] S. Noikajana and T. Suwannasart, "An improved test case generation method for web service testing from WSDL-S and OCL with pair-wise testing technique," in *Proc. of COMPSAC'09*, vol. 1, 2009, pp. 115–123.

[294] T. C. Norman, C. Bountra, A. M. Edwards, K. R. Yamamoto, and S. H. Friend, "Leveraging crowdsourcing to facilitate the discovery of new medicines," *Science Translational Medicine*, vol. 3, no. 88, 2011.

[295] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *Proc. of ESEM'13*, 2013, pp. 55–64.

[296] F. S. Ocariza Jr, "Characterizing the javascript errors that occur in production web applications," Ph.D. dissertation, University of British Columbia, 2012.

[297] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proc. of ISSRE'11*, 2011, pp. 100–109.

[298] N. Olofsson, "Automated testing of a dynamic web application," Master's thesis, Linköping University, June 2014.

[299] D. L. Olson and K. Rosacker, "Crowdsourcing and open source software participation," *Service Business*, vol. 7, no. 4, pp. 499–511, November 2012.

[300] M. Orlov and M. Sipper, "Flight of the FINCH through the java wilderness," *IEEE Transactions Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, 2011.

[301] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, Mar. 2013.

[302] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proc. of GECCO'13*, 2013, pp. 1461–1468.

[303] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," *Proc. of ICSE'07*, pp. 75–84, May 2007.

[304] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *Proc. of RE'13*, July 2013, pp. 125–134.

[305] S. C. J. Palvia and S. S. Sharma, "E-government and e-governance: definitions/-domain framework and status around the world," in *Proc. of ICEG'07*, 2007.

[306] D. Papamartzivanos, D. Damopoulos, and G. Kambourakis, "A cloud-based architecture to crowdsource mobile app privacy leaks," in *Proc. of PCI'14*, 2014, pp. 59:1–59:6.

[307] C. Parnin, C. Treude, L. Grammel, and M. Storey, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," Georgia Institute of Technology, Tech. Rep., 2012.

[308] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the crowd solve the oracle problem?" in *Proc. of ISSTA'13*, March 2013, pp. 342–351.

[309] A. Pawlik, J. Segal, M. Petre, and H. Sharp, "Crowdsourcing scientific software documentation: a case study of the NumPy documentation project," *Computing in Science and Engineering*, 2014.

[310] O. Pedreira, F. García, N. Brisaboa, and M. Piattini, "Gamification in soft-

ware engineering–a systematic mapping," *Information and Software Technology*, vol. 57, pp. 157–168, 2015.

[311] X. Peng, M. Ali Babar, and C. Ebert, "Collaborative Software Development Platforms for Crowdsourcing," *IEEE Software*, vol. 31, no. 2, pp. 30–36, 2014.

[312] D. Peters and D. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, Mar 1998.

[313] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, pp. 901–924, 2015.

[314] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement & code transplants to specialise a C++ program to a problem class," in *Proc. of EuroGP'14*, Granada, Spain, April 2014, pp. 132–143.

[315] D. Phair, "Open crowdsourcing: Leveraging community software developers for IT projects," PhD. in Computer Sci., Colorado Technical University, 2012.

[316] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Proc. of ICSE'13*, 2013, pp. 112–121.

[317] R. Pham, L. Singer, and K. Schneider, "Building test suites in social coding sites by leveraging drive-by commits," in *Proc. of ICSE'12*, May 2013, pp. 1209–1212.

[318] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Prompter: A self-confident recommender system," in *Proc. of ICSME'14*, Sept 2014, pp. 577–580.

[319] L. Ponzanelli, "Exploiting crowd knowledge in the IDE," Master's thesis, University of Lugano, 2012.

[320] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *Proc. of CSMR'13*, March 2013, pp. 57–66.

[321] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *Proc. of ICSE'13*, May 2013, pp. 1295–1298.

[322] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proc. of*

*MSR'14*, 2014, pp. 102–111.

[323] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *Proc. of ICST'10*, April 2010, pp. 132–141.

[324] R. Prikladnicki, L. Machado, E. Carmel, and C. R. B. de Souza, "Brazil software crowdsourcing: A first step in a multi-year study," in *Proc. of CSISE'14*, June 2014, pp. 1–4.

[325] S. Ramakrishnan and V. Srinivasaraghavan, "Delivering software projects using captive university crowd," in *Proc. of CHASE'14*, 2014, pp. 115–118.

[326] J. Ross, A. Zaldivar, L. Irani, and B. Tomlinson, "Who are the Turkers? worker demographics in Amazon mechanical turk," Department of Informatics, University of California, Irvine, USA, Tech. Rep., 2009.

[327] J. Ross, L. Irani, M. Silberman, A. Zaldivar, and B. Tomlinson, "Who are the crowdworkers? shifting demographics in mechanical turk," in *Proc. of CHI'10*, 2010, pp. 2863–2872.

[328] M. Saengkhattiya, M. Sevandersson, and U. Vallejo, "Quality in crowdsourcing - How software quality is ensured in software crowdsourcing," Master's thesis, Lund University, 2012.

[329] K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller, "Using dynamic symbolic execution to generate inputs in search-based gui testing," in *Proc. of SBST'15*, 2015, pp. 32–35.

[330] J. Saxe, R. Turner, and K. Blokhin, "CrowdSource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model," in *Proc. of MALWARE'14*, 2014, pp. 68–75.

[331] G. D. Saxton, O. Oh, and R. Kishore, "Rules of Crowdsourcing: Models, Issues, and Systems of Control," *Information Systems Management*, vol. 30, no. 1, pp. 2–20, January 2013.

[332] T. W. Schiller and M. D. Ernst, "Reducing the barriers to writing verified specifications," in *Proc. of OOPSLA'12*, 2012, pp. 95–112.

[333] T. W. Schiller, "Reducing the usability barrier to specification and verification," Ph.D. dissertation, University of Washington, 2014.

[334] C. Schneider and T. Cheung, "The power of the crowd: Performing usability testing using an on-demand workforce," in *Proc. of ICISD'11*, 2011.

[335] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. Jones, Eds., 2006, vol. 4144, pp. 419–423.

[336] N. Seyff, F. Graf, and N. Maiden, "Using mobile RE tools to give end-users their own voice," in *Proc. of RE'10*, 2010, pp. 37–46.

[337] N. Shah, A. Dhanesha, and D. Seetharam, "Crowdsourcing for e-Governance: Case study," in *Proc. of ICEGOV'09*, 2009, pp. 253–258.

[338] H. Shahriar and M. Zulkernine, "MUTEC: Mutation-based testing of cross site scripting," in *Proc. of SESS'09*, May 2009, pp. 47–53.

[339] M. Sharifi, E. Fink, and J. G. Carbonell, "SmartNotes: Application of crowdsourcing to the detection of web threats," in *Proc. of SMC'16*, 2011, pp. 1346–1350.

[340] H. Sharp, N. Baddoo, S. Beecham, T. Hall, and H. Robinson, "Models of motivation in software engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 219–233, 2009.

[341] N. Sherief, "Software evaluation via users' feedback at runtime," in *Proc. of EASE'14*, 2014, pp. 1–4.

[342] N. Sherief, N. Jiang, M. Hosseini, K. Phalp, and R. Ali, "Crowdsourcing software evaluation," in *Proc. of EASE'14*, 2014, pp. 1–4.

[343] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, April 2008.

[344] H. A. Simon, *The New Science of Management Decision.* Harper & Brothers, 1960.

[345] R. Snijders, "Crowd-centric requirements engineering: A method based on crowdsourcing and gamification," Master's thesis, Utrecht University, 2015.

[346] R. Snijders and F. Dalpiaz, "Crowd-centric requirements engineering," in *Proc. of CGCloud'14*, 2014.

[347] D. Sobel, *Longitude: The true story of a lone genius who solved the greatest scientific problem of his time.* New York: Walker, 1995.

[348] G. Standish, "The chaos report," http://www.standishgroup.com/sample_research_files/chaos_report_1994.pdf, 1994, Accessed: 2015-01-27.

[349] O. Starov, "Cloud platform for research crowdsourcing in mobile testing," Mas-

ter's thesis, East Carolina University, 2013.

[350] Statista, "Number of apps available in leading app stores as of june 2016," https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores, 2016.

[351] K.-J. Stol and B. Fitzgerald, "Two's company, three's a crowd: A case study of crowdsourcing software development," in *Proc. of ICSE'14*, 2014, pp. 187–198.

[352] K.-j. Stol and B. Fitzgerald, "Research protocol for a case study of crowdsourcing software development," Available from: http://staff.lero.ie/stol/publications, University of Limerick, 2014.

[353] K.-J. Stol and B. Fitzgerald, "Researching crowdsourcing software development: Perspectives and concerns," in *Proc. of CSISE'14*, 2014, pp. 7–10.

[354] K. T. Stolee and S. Elbaum, "Exploring the use of crowdsourcing to support empirical studies in software engineering," in *Proc. of ESEM'10*, 2010, pp. 1–4.

[355] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 3, pp. 26:1–26:45, June 2014.

[356] K. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1654–1679, Dec 2013.

[357] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *Proc. of FoSER'10*, ser. FoSER '10, 2010, pp. 359–364.

[358] H. Tajedin and D. Nevo, "Determinants of success in crowdsourcing software development," in *Proc. of CPR'13*, 2013, pp. 173–178.

[359] H. Tajedin and D. Nevo, "Value-adding intermediaries in software crowdsourcing," in *Proc. of HICSS'14*, January 2014, pp. 1396–1405.

[360] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proc. of ICSE'16*, 2016, pp. 321–332.

[361] A. Teinum, "User testing tool towards a tool for crowdsource-enabled accessibility evaluation of websites," Master's thesis, University of Agder, 2013.

[362] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided

test generation for web applications," in *Proc. of ICSE'13*, 2013.

[363] N. Tillmann and J. de Halleux, "Pex-White Box Test Generation for .NET," in *Proc. of TAP'08*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hhnle, Eds., 2008, vol. 4966, pp. 134–153.

[364] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich, "TouchDevelop: Programming cloud-connected mobile devices via touchscreen," in *Proc. of ONWARD'11*, 2011, pp. 49–60.

[365] L. Tran-Thanh, S. Stein, A. Rogers, and N. R. Jennings, "Efficient crowdsourcing of unknown experts using bounded multi-armed bandits," *Artificial Intelligence*, vol. 214, pp. 89–111, September 2014.

[366] W.-T. Tsai, W. Wu, and M. N. Huhns, "Cloud-based software crowdsourcing," *IEEE Internet Computing*, vol. 18, no. 3, pp. 78–83, May 2014.

[367] Y.-H. Tung and S.-S. Tseng, "A novel approach to collaborative testing in a crowdsourcing environment," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2143–2153, August 2013.

[368] Y. Usui and S. Morisaki, "An Approach for Crowdsourcing Software Development," *Proc. of IWSM/MENSURA'11*, pp. 32–33, 2011.

[369] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[370] L. R. Varshney, "Participation in crowd systems," in *Proc. of Allerton'12*, October 2012, pp. 996–1001.

[371] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between software development and crowdsourced knowledge," in *Proc. of SocialCom'13*, September 2013, pp. 188–195.

[372] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," *Proc. of CSCW'14*, pp. 342–354, 2014.

[373] V. Veerappa and E. Letier, "Understanding clusters of optimal solutions in multi-objective decision problems," in *Proc. of RE'11*, 2011, pp. 89–98.

[374] M. Visconti and C. Cook, "An overview of industrial software documentation practice," in *Proc. of SCCC'02*, 2002, pp. 179–186.

[375] R. Vliegendhart, E. Dolstra, and J. Pouwelse, "Crowdsourced user interface testing for multimedia applications," in *Proc. of CrowdMM'12*, 2012, pp. 21–22.

[376] W3C, "UI events specification," http://www.w3.org/TR/uievents, 2015.

[377] H. Wang, Y. Wang, and J. Wang, "A participant recruitment framework for crowdsourcing based software requirement acquisition," in *Proc. of ICGSE'14*, August 2014, pp. 65–73.

[378] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Proc. of ESEC/FSE'13*, August 2013, pp. 455–465.

[379] J. Warner, "Next steps in e-government crowdsourcing," in *Proc. of dg.o'11*, 2011, pp. 177–181.

[380] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proc. of FoSER'10*, 2010, pp. 397–400.

[381] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proc. of ISSTA'08*, 2008, pp. 249–260.

[382] R. Watro, K. Moffitt, T. Hussain, D. Wyschogrod, J. Ostwald, D. Kong, C. Bowers, E. Church, J. Guttman, and Q. Wang, "Ghost Map: Proving software correctness using games," in *Proc. of SECURWARE'14*, 2014.

[383] C. Watson, F. W. B. Li, and J. L. Godwin, "BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair," in *Proc. of ICWL'12*, 2012, pp. 228–239.

[384] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, November 1982.

[385] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, 2011.

[386] D. Wightman, "Search interfaces for integrating crowdsourced code snippets within development environments," Ph.D. dissertation, Queen's University, 2013.

[387] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, no. 6, pp. 80–83, 1945.

[388] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization

to software testing," in *Proc. of. GECCO'07*, 2007, pp. 1121–1128.

[389] F. Wu, M. Harman, Y. Jia, J. Krinke, and W. Weimer, "Deep parameter optimisation," in *Proc. of GECCO'15*, Madrid, Spain, July 2015, pp. 1375–1382.

[390] W. Wu, W.-T. Tsai, and W. Li, "An evaluation framework for software crowdsourcing," *Frontiers of Computer Science*, vol. 7, no. 5, pp. 694–709, August 2013.

[391] W. Wu, W. T. Tsai, and W. Li, "Creative software crowdsourcing: from components and algorithm development to project concept formations," *International Journal of Creative Computing*, vol. 1, no. 1, pp. 57–91, 2013.

[392] L. Xiao and H.-Y. Paik, "Supporting complex work in crowdsourcing platforms: A view from service-oriented computing," in *Proc. of ASWEC'14*, April 2014, pp. 11–14.

[393] T. Xie, "Cooperative testing and analysis: Human-tool, tool-tool, and human-human cooperations to get work done," in *Proc. of SCAM'12 (Keynote)*, 2012.

[394] T. Xie, J. Bishop, R. N. Horspool, N. Tillmann, and J. de Halleux, "Crowdsourcing code and process via Code Hunt," in *Proc. of CSISE'15*, May 2015.

[395] X. L. Xu and Y. Wang, "On the Process Modeling of Software Crowdsourcing Based on Competitive Relation," *Advanced Materials Research*, vol. 989-994, pp. 4708–4712, July 2014.

[396] X. L. Xu and Y. Wang, "Crowdsourcing Software Development Process Study on Ultra-Large-Scale System," *Advanced Materials Research*, vol. 989-994, pp. 4441–4446, July 2014.

[397] H. Xue, "Using redundancy to improve security and testing," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2013.

[398] M. Yan, H. Sun, and X. Liu, "iTest: Testing software with mobile crowdsourcing," in *Proc. of CrowdSoft'14*, 2014, pp. 19–24.

[399] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proc. of FASE'13*, 2013, pp. 250–265.

[400] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. of ISSTA'07*, 2007, pp. 140–150.

[401] S. Yoo, M. Harman, and S. Ur, "GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards," *Journal of*

*Empirical Software Engineering*, vol. 18, no. 3, pp. 550–593, June 2013.

[402] M.-C. Yuen, I. King, and K.-S. Leung, "A survey of crowdsourcing systems," in *Proc. of SocialCom'11*, Oct 2011, pp. 766–773.

[403] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using social media for code recommendation," in *Proc. of RSSE'12*, June 2012, pp. 38–42.

[404] O. F. Zaidan and C. Callison-Burch, "Crowdsourcing translation: Professional quality from non-professionals," in *Proc. of ACL'11*, 2011, pp. 1220–1229.

[405] Y. Zhang, M. Harman, and S. A. Mansouri, "The multi-objective next release problem," in *Proc. of GECCO'07*, 2007, pp. 1129–1137.

[406] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proc. of WWW'11*, 2011, pp. 805–814.

[407] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[408] S. Zogaj and U. Bretschneider, "Crowdtesting with testcloud - managing the challenges of an intermediary in a crowdsourcing business model," in *Proc. of ECIS'13*, 2013, p. 143.

[409] S. Zogaj, U. Bretschneider, and J. M. Leimeister, "Managing crowdsourced software testing: A case study based insight on the challenges of a crowdsourcing intermediary," *Journal of Business Economics*, vol. 84, no. 3, pp. 375–405, 2014.